

**Laporan Tugas Besar 1**  
**IF3070 Dasar Inteligensi Artifisial**  
**Pencarian Solusi Diagonal Magic Cube dengan Local Search**



**Disusun oleh:**

**Kelompok 20**

Muhammad Daffa Kusuma Atmaja	18222108
Ahmad Fawwazi	18222117
Muhammad Faishal Putra	18222129
Muhammad Faishal Firdaus	18222136

**Program Studi Sistem dan Teknologi Informasi Sekolah Teknik  
Elektro dan Informatika Institut Teknologi Bandung**

## **DAFTAR ISI**

I. Deskripsi Persoalan.....	3
II. Pembahasan.....	4
A. Pemilihan objective function.....	4
B. Penjelasan implementasi algoritma local search (berisi deskripsi fungsi/kelas beserta source codenya).....	12
C. Hasil eksperimen dan analisis (disertai dengan visualisasi dari program yang telah dibuat)...	27
III. Kesimpulan dan Saran.....	63
IV. Pembagian tugas tiap anggota kelompok.....	64
V. Referensi.....	65

## I. Deskripsi Persoalan

	25	16	80	104	90	
	115	98	4	1	97	90
	42	111	85	2	75	97
	66	72	27	102	48	75
	67	18	119	106	5	13
67	18	119	106	5	5	86
116	17	14	73	95	95	94
40	50	81	65	79	79	10
56	120	55	49	35	35	59
36	110	46	22	101	101	60

Pada tugas besar 1 ini, permasalahan yang dihadapi adalah menyelesaikan Diagonal Magic Cube dengan ukuran  $5 \times 5 \times 5$  menggunakan algoritma *local search*. Diagonal Magic Cube adalah sebuah kubus yang tersusun dari angka-angka 1 hingga  $n^3$  (dalam kasus ini, 1 hingga  $5^3$  atau sama dengan 125) tanpa pengulangan, di mana  $n$  adalah panjang sisi kubus. Permasalahan ini bertujuan untuk menemukan susunan angka di dalam kubus tersebut sehingga memenuhi kondisi magic number.

Magic number pada kubus ini didefinisikan sebagai jumlah angka yang sama di seluruh baris, kolom, tiang, serta diagonal ruang dan bidang dari kubus. Adapun properti-properti yang harus dipenuhi oleh solusi Diagonal Magic Cube adalah sebagai berikut:

1. Jumlah angka-angka pada setiap baris sama dengan magic number.
2. Jumlah angka-angka pada setiap kolom sama dengan magic number.
3. Jumlah angka-angka pada setiap tiang sama dengan magic number.
4. Jumlah angka-angka pada seluruh diagonal ruang sama dengan magic number.

5. Jumlah angka-angka pada seluruh diagonal pada potongan bidang dari kubus sama dengan magic number.

Tugas ini berfokus pada penerapan algoritma *local search* untuk mencari solusi yang memenuhi semua kondisi tersebut. Dalam implementasinya, algoritma akan memulai dari susunan angka acak pada kubus, kemudian pada setiap iterasi, dua angka di dalam kubus akan ditukar posisinya untuk memperbaiki solusi menuju solusi yang memenuhi kondisi magic number. Algoritma yang digunakan dalam tugas ini mencakup beberapa variasi *local search*, seperti Steepest Ascent Hill-climbing, Stochastic Hill-climbing, Simulated Annealing, dan Genetic Algorithm.

Tujuan akhir dari permasalahan ini adalah menemukan solusi Diagonal Magic Cube yang valid melalui penggunaan algoritma *local search*, serta mengevaluasi performa masing-masing algoritma dalam menyelesaikan permasalahan ini.

## II. Pembahasan

### A. Pemilihan objective function

Objective function yang dipilih dalam tugas besar ini bertujuan untuk mengukur seberapa dekat suatu susunan *Diagonal Magic Cube* berukuran 5x5x5 memenuhi kriteria sebagai solusi yang valid dengan *magic number* tertentu. Pada Magic Cube dengan ukuran  $n \times n \times n$ , rumus umum yang digunakan untuk magic number adalah:

$$M = \frac{n(n^3 + 1)}{2}$$

Dengan  $n$  adalah panjang sisi kubus. Dalam kasus ini,  $n = 5$ , sehingga magic number  $M$  adalah:

$$M = \frac{5(5^3 + 1)}{2} = \frac{5(125 + 1)}{2} = 315$$

Setelah Magic Number diketahui, terdapat hal lain dari objektif kali ini yang perlu dicari, yaitu banyaknya garis - garis lurus yang hasil penjumlahannya harus sebesar 315 (Magic Number). Garis - garis lurus pada Magic Cube ini berupa baris, kolom, tiang, diagonal ruang (triagonal), dan diagonal pada potongan bidang. Pada Magic Cube Walter Trumps didapat detail total untuk masing - masing garis lurus pada kubus tersebut sebagai berikut:

- 5 baris pada setiap level
- 5 Kolom pada setiap level
- 2 diagonal pada setiap level vertikal ataupun horizontal
- 25 tiang
- 4 triagonal

Total banyak garis ini bisa dicari baik secara manual dengan menggunakan detail diatas ataupun menggunakan persamaan pada tulisan Walter Trump pada bagian sebelumnya:

- Pencarian banyaknya garis lurus dengan persamaan pada tulisan Walter Trump.

Pada tulisan Walter Trump di bab sebelumnya, terdapat suatu persamaan untuk mencari banyaknya garis lurus. Tulisan tersebut berupa

*“...There are  $3m^2+6m+4$  straight lines:  $m^2$  rows,  $m^2$  columns,  $m^2$  pillars,  $6m$  diagonals (= short diagonals) and 4 triagonals (= long diagonals). ...”*

Maka dapat disimpulkan bahwa banyaknya garis lurus yang harus memenuhi Magic Number dapat dicari dengan menggunakan persamaan sebagai berikut.

$$l = 3m^2 + 6m + 4$$

$$m = \text{ordo kubus}$$

$$l = \text{Banyaknya garis lurus atau straight lines}$$

Diketahui ordo kubus yang digunakan berupa ordo 5 sehingga didapat banyaknya garis lurus pada kubus ini sebagai berikut.

$$l = 3 \times 5^2 + 6 \times 5 + 4 = 109$$

Maka, banyaknya garis lurus pada Magic Cube ordo 5 adalah 109 garis.

- Pencarian banyaknya garis lurus dengan perhitungan manual berdasarkan informasi yang didapat

Pada detail sebelumnya terdapat detail informasi mengenai banyaknya masing - masing straight lines. Dengan informasi tersebut, akan dilakukan perhitungan manual sebagai berikut.

- 5 baris pada setiap level. *Total baris* =  $5 \times 5 = 25$
- 5 Kolom pada setiap level. *Total kolom* =  $5 \times 5 = 25$
- 2 diagonal pada setiap level vertikal ataupun horizontal.  
*Total diagonal bidang* =  $2 \times 5 \times 3 = 30$
- 25 tiang. *Total tiang* = 25
- 4 triagonal. *Total triagonal* = 4

Maka, didapat banyaknya garis lurus pada Magic Cube ordo 5 sebagai berikut.

$$l = 25 + 25 + 30 + 25 + 4 = 109$$

Kedua cara tersebut (perhitungan manual berdasarkan informasi yang didapat dan persamaan dari Walter Trump) menghasilkan banyaknya garis lurus yang sama yaitu 109. Maka dapat dipastikan bahwa banyaknya garis lurus pada Magic Cube ordo 5 ini berjumlah 109 garis.

Dalam tugas besar kali ini, kami mengimplementasikan objective function pada fungsi evaluate(cubeArray), fungsi ini pertama-tama menghitung jumlah

angka pada setiap baris dan kolom dalam setiap layer kubus, yang mencakup bidang XY-plane. Jika jumlah pada baris atau kolom sama dengan *magic number*, garis tersebut dianggap sesuai dan dihitung dalam variabel matchingLines. Selanjutnya, fungsi ini memeriksa tiang-tiang vertikal yang terbentuk ketika menelusuri setiap kolom pada layer yang berbeda dalam kubus. Garis-garis vertikal ini menghubungkan semua layer dalam kubus dan juga perlu mencapai jumlah *magic number* untuk dianggap sesuai.

Selain baris, kolom, dan tiang, objective function ini juga memeriksa diagonal ruang tiga dimensi (atau *space diagonals*), yaitu empat garis yang melintang dari satu sudut kubus ke sudut lainnya dalam ruang tiga dimensi. Jika jumlah elemen pada setiap diagonal ruang ini sama dengan *magic number*, diagonal tersebut dihitung sebagai garis yang memenuhi syarat. Terakhir, fungsi ini juga menghitung diagonal pada setiap bidang atau layer di dalam kubus, yang disebut sebagai diagonal bidang atau *face diagonals*. Diagonal ini mencakup diagonal dalam XY-plane, XZ-plane, dan YZ-plane pada setiap layer. Setiap kali diagonal bidang ini mencapai jumlah *magic number*, fungsi menambahkannya sebagai garis yang sesuai atau matchingLines.

Hasil akhir dari fungsi ini adalah jumlah total *matchingLines*, atau garis-garis yang telah memenuhi kriteria *magic number*. Objective function ini menggambarkan seberapa optimal susunan angka dalam kubus untuk memenuhi kondisi sebagai *Diagonal Magic Cube*. Dengan mengoptimalkan objective function ini, algoritma *local search* yang digunakan dalam tugas besar akan berusaha meningkatkan jumlah *matchingLines* pada setiap iterasi, mendekati konfigurasi Magic Cube yang memenuhi syarat optimal. Berikut merupakan potongan kode dari objective function:

```
function evaluate(cubeArray) {  
    let matchingLines = 0;  
  
    // Fungsi untuk mengakses elemen dalam array 1D sesuai layer, baris, dan  
    kolumn
```

```

function getValue(layer, row, col) {
    return cubeArray[layer * 25 + row * 5 + col];
}

// Periksa semua baris dan kolom pada setiap layer
for (let layer = 0; layer < size; layer++) {
    for (let i = 0; i < size; i++) {
        // Hitung jumlah pada setiap baris
        let rowSum = 0;
        for (let j = 0; j < size; j++) {
            rowSum += getValue(layer, i, j);
        }
        if (rowSum === magicNumber) matchingLines++;
    }
}

// Hitung jumlah pada setiap kolom
let colSum = 0;
for (let j = 0; j < size; j++) {
    colSum += getValue(layer, j, i);
}
if (colSum === magicNumber) matchingLines++;

}

}

// Periksa tiang (baris vertikal melalui layer-layer)
for (let i = 0; i < size; i++) {
    for (let j = 0; j < size; j++) {
        let pillarSum = 0;
        for (let layer = 0; layer < size; layer++) {
            pillarSum += getValue(layer, i, j);
        }
        if (pillarSum === magicNumber) matchingLines++;
    }
}

// Periksa empat triagonal (diagonal ruang 3D)
let spaceDiagonal1 = 0;
let spaceDiagonal2 = 0;
let spaceDiagonal3 = 0;
let spaceDiagonal4 = 0;

for (let i = 0; i < size; i++) {
    spaceDiagonal1 += getValue(i, i, i);
    spaceDiagonal2 += getValue(i, i, size - i - 1);
}

```

```

        spaceDiagonal3 += getValue(i, size - i - 1, i);
        spaceDiagonal4 += getValue(i, size - i - 1, size - i - 1);
    }
    if (spaceDiagonal1 === magicNumber) matchingLines++;
    if (spaceDiagonal2 === magicNumber) matchingLines++;
    if (spaceDiagonal3 === magicNumber) matchingLines++;
    if (spaceDiagonal4 === magicNumber) matchingLines++;

    for (let layer = 0; layer < size; layer++) {
        // Face diagonals pada XY-plane (per layer Z)
        let faceDiagonalXY1 = 0;
        let faceDiagonalXY2 = 0;
        for (let i = 0; i < size; i++) {
            faceDiagonalXY1 += getValue(layer, i, i);
            faceDiagonalXY2 += getValue(layer, i, size - i - 1);
        }
        if (faceDiagonalXY1 === magicNumber) matchingLines++;
        if (faceDiagonalXY2 === magicNumber) matchingLines++;

        // Face diagonals pada XZ-plane (per layer Y)
        let faceDiagonalXZ1 = 0;
        let faceDiagonalXZ2 = 0;
        for (let i = 0; i < size; i++) {
            faceDiagonalXZ1 += getValue(i, layer, i);
            faceDiagonalXZ2 += getValue(i, layer, size - i - 1);
        }
        if (faceDiagonalXZ1 === magicNumber) matchingLines++;
        if (faceDiagonalXZ2 === magicNumber) matchingLines++;

        // Face diagonals pada YZ-plane (per layer X)
        let faceDiagonalYZ1 = 0;
        let faceDiagonalYZ2 = 0;
        for (let i = 0; i < size; i++) {
            faceDiagonalYZ1 += getValue(i, i, layer);
            faceDiagonalYZ2 += getValue(size - i - 1, i, layer);
        }
        if (faceDiagonalYZ1 === magicNumber) matchingLines++;
        if (faceDiagonalYZ2 === magicNumber) matchingLines++;
    }
    return matchingLines;
}
}

```

## **B. Penjelasan implementasi algoritma local search (berisi deskripsi fungsi/kelas beserta source codenya)**

### **1) Steepest Ascent Hill Climbing**

Steepest Ascent Hill Climbing merupakan salah satu jenis algoritma local search Hill Climbing. Algoritma ini akan mengevaluasi semua neighbor dari solusi yang dimiliki saat ini, kemudian Steepest Ascent akan memilih neighbor yang memiliki nilai heuristik yang tertinggi dan menjadikan neighbor tersebut sebagai current state. Secara garis besar, berikut alur algoritma Steepest Ascent Hill Climbing:

- Inisiasi populasi awal
- Perhitungan nilai objective function dari inisiasi awal
- Penentuan max iterasi
- Loop dimulai dengan pencarian semua neighbor
- Mengevaluasi nilai objective function dari setiap neighbor yang dihasilkan
- Memilih neighbor terbaik dengan nilai objective function yang lebih baik daripada current state untuk menjadi current state yang baru
- Looping Akan Berhenti Jika Telah Mencapai Max Iterasi atau Local Optimum

Dalam tugas ini, pengimplementasian algoritma Steepest Ascent Hill menggunakan function sebagai berikut:

- `steepestAscentHillClimbing(initialCubeArray)`

Fungsi ini bertanggung jawab mengimplementasikan algoritma Steepest Ascent Hill Climbing. Algoritma ini dimulai dengan menginisialisasi current state dari `initialCubeArray`, menghitung nilai objective function awal dengan `currentScore`. Pada setiap iterasi, algoritma menghasilkan semua neighbor dengan memanggil fungsi `steepestGenerateRandomNeighbor`. Algoritma

kemudian memilih tetangga dengan nilai objective function terbaik. Jika tetangga memiliki nilai yang lebih baik daripada current state, algoritma akan berpindah ke tetangga tersebut. Jika tidak, maka algoritma akan berhenti pada local optimum. Berikut merupakan potongan code dari fungsi steepestAscentHillClimbing(initialCubeArray):

```
function steepestAscentHillClimbing(initialCubeArray)
{
    let current_state = [...initialCubeArray];
    let current_obj = evaluate(current_state); // Menghitung nilai objective
    function pada current state
    let iterasi = 0;
    const max_iterasi = 1000;
    const scores = [current_obj];

    const steepeststartTime = Date.now();

    while (iterasi < max_iterasi)
    {
        iterasi = iterasi + 1;

        // menghasilkan semua neighbor yang ada
        const neighbors = steepestgenerateRandomNeighbor(current_state);

        // mencari neighbor dengan score terbaik
        let best_neighbor = null;
        let best_score = current_obj;

        for (const neighbor of neighbors)
        {
            const neighborScore = evaluate(neighbor);
            if (neighborScore > best_score)
            {
                best_score = neighborScore;
                best_neighbor = neighbor;
            }
        }

        // Jika ada neighbor dengan objective function lebih baik, current
        state pindah ke neighbor tersebut
    }
}
```

```

        if (best_neighbor !== null)
        {
            current_state = best_neighbor;
            current_obj = best_score;

            cubeNumbers = [...current_state];
            renderNumbers();
        }
        else
        {
            // jika tidak ada neighbor lebih baik, berhenti iterasi
            console.log("Local optimum tercapai pada: ", iterasi);
            break;
        }
        // menyimpan nilai objective function
        scores.push(current_obj);
    }

    const steepestendTime = Date.now();

    console.log("Iterasi final pada: ", iterasi);

    return {
        initialState: initialCubeArray,
        finalState: current_state, // current state final
        finalScore: current_obj, // Nilai objective function akhir
        scores, // Semua skor untuk iterasi
        duration: (steepestendTime - steepeststartTime) / 1000,
        iteration: iterasi // Total iterasi yang dijalankan
    };
}

```

- `steepestgenerateRandomNeighbor(CubeArray)`

Fungsi ini digunakan untuk menghasilkan semua neighbor dari state saat ini. Setelah semua kemungkinan neighbor dihasilkan, fungsi ini mengembalikan array yang berisi neighbor tersebut. Berikut merupakan potongan code dari fungsi `steepestgenerateRandomNeighbor(CubeArray)`:

```
function steepestgenerateRandomNeighbor(cubeArray) {
```

```

const neighbors = [];

// Iterasi melalui semua pasangan elemen
for (let i = 0; i < cubeArray.length; i++) {
    for (let j = i + 1; j < cubeArray.length; j++) {
        const newNeighbor = [...cubeArray];
        [newNeighbor[i], newNeighbor[j]] = [newNeighbor[j],
newNeighbor[i]]; // Tukar elemen
        neighbors.push(newNeighbor); // Tambahkan ke daftar neighbor
    }
}

return neighbors;
}

```

## 2) Stochastic Hill Climbing

Stochastic hill climbing adalah salah satu jenis algoritma local search hill climbing. Perbedaan algoritma ini dengan hill climbing yang biasa adalah ketika pemilihan neighbor baru dilakukan secara acak, dan ketika neighbor baru tersebut memiliki nilai value yang lebih tinggi, maka current state akan berpindah. Untuk alur algoritma ini adalah sebagai berikut.

- Inisiasi state awal magic cube
- Perhitungan nilai objective function dari inisiasi awal
- Penentuan max iterasi
- Loop dimulai dengan pencarian random neighbor dengan melakukan penukaran 2 value pada bagian magic cube secara acak.
- Algoritma akan mengecek nilai objective function dari neighbor ini, jika lebih baik maka current state akan berpindah, jika tidak maka algoritma akan terus melakukan looping.
- Looping akan berhenti jika telah mencapai max iterasi.

Implementasi algoritma stochastic hill climbing ini memerlukan function berikut.

- Random\_Neighbor(cubeArray)

Function ini dilakukan untuk melakukan pencarian neighbor secara random. Function ini akan menerima array yang berisi setiap value dari magic cube. Lalu function akan melakukan penukaran 2 dari value magic cube dengan menggunakan index-nya secara acak (menggunakan `Math.random()` pada `length` dari array untuk mendapatkan index secara acak). Setelah itu function akan mengembalikan array cube yang berisi 2 value yang telah ditukar secara acak.

```
function Random_Neighbor(cubeArray) {  
  
    const New_Neighbor = [...cubeArray];  
    const i = Math.floor(Math.random() * cubeArray.length);  
    const j = Math.floor(Math.random() * cubeArray.length);  
  
    [New_Neighbor[i], New_Neighbor[j]] = [New_Neighbor[j], New_Neighbor[i]]; //  
    menukar menggunakan index  
  
    return New_Neighbor;  
}
```

### 3) Simulated Annealing

Simulated Annealing adalah algoritma metaheuristik yang diadaptasi dari proses fisika di mana material didinginkan secara perlahan untuk mencapai keadaan energi yang lebih rendah atau stabil. Algoritma ini digunakan untuk mencari solusi optimal dalam ruang solusi yang kompleks dengan menghindari perangkap local optimum melalui pendekatan probabilistik. Pada dasarnya alur pada algoritma ini adalah sebagai berikut :

- Inisiasi state awal magic cube dan tetapkan suhu awal yang tinggi
- Hitung nilai *objective function* dari solusi awal.
- Tetapkan *cooling rate*, suhu minimum, dan jumlah iterasi maksimum.

- Pilih neighbor (solusi tetangga) dari solusi saat ini secara acak.
- Hitung nilai *objective function* untuk neighbor yang dipilih.
- Jika neighbor lebih baik, terima sebagai solusi baru.
- Jika neighbor lebih buruk, terima berdasarkan probabilitas

$$P(\text{accept}) = e^{\Delta E/T}$$

- Kurangi suhu dengan *cooling rate* setelah setiap iterasi.
- Ulangi langkah 4-8 hingga mencapai iterasi maksimum atau suhu minimum.
- Hentikan algoritma jika solusi optimal ditemukan atau kondisi berhenti tercapai.

Dalam tugas ini, pengimplementasian algoritma *Simulated Annealing* menggunakan fungsi berikut:

- `simulatedAnnealing(initialCubeArray)`

Algoritma ini dimulai dengan menginisialisasi current state dari *initialCubeArray* dan menghitung nilai objective function awal dengan *currentScore*. Pada setiap iterasi, algoritma memilih satu neighbor secara acak dengan memanggil fungsi *generateRandomNeighbor*.

Jika neighbor memiliki nilai objective function yang lebih baik, algoritma akan menerima neighbor tersebut sebagai current state baru. Namun, jika neighbor memiliki nilai yang lebih buruk, algoritma masih memiliki peluang untuk menerima neighbor tersebut berdasarkan probabilitas yang dihitung dari persamaan eksponensial ( $e^{\Delta E/T}$ ), dimana  $\Delta E$  adalah perbedaan antara nilai objective function neighbor dan current state, dan T adalah suhu saat ini.

Suhu (T) akan dikurangi setelah setiap iterasi dengan faktor cooling rate sampai mencapai batas minimum atau iterasi maksimum. Ketika suhu turun ke nilai yang sangat kecil, algoritma menjadi lebih ketat dalam menerima solusi yang lebih buruk, sehingga proses pencarian cenderung berhenti pada solusi terbaik yang telah ditemukan. Jika neighbor lebih buruk diterima, algoritma menganggap kondisi "stuck" atau terjebak dalam local optimum.

Secara garis besar, berikut adalah potongan kode dari fungsi *simulatedAnnealing*:

```
function simulatedAnnealing(initialCubeArray) {
    let currentState = [...initialCubeArray];
    let currentScore = evaluate(currentState);
    let temperature = 10000;
    const coolingRate = 0.999;
    const maxIterations = 50000;
    const scores = [currentScore];
    const deltaETValues = [];
    const stuckThreshold = 50;
    let stuckCounter = 0;
    const stuckIterations = [];
    const optimalScore = 109;

    let iteration = 0;
    const simulatedannealingStartTime = Date.now();

    if (currentScore === optimalScore) {
        console.log("Cube adalah magic cube sempurna dari awal jadi tidak usah dicari lagi.");
        return {
            finalState: currentState,
            finalScore: currentScore,
            scores,
            deltaETValues,
            stuckIterations,
            duration: 0,
            iteration: 0
    
```

```

        };

    }

    while (iteration < maxIterations) {
        iteration++;

        const randomNeighbor = generateRandomNeighbor(currentState);
        const neighborScore = evaluate(randomNeighbor);
        const deltaE = neighborScore - currentScore;
        const deltaET = Math.exp(deltaE / temperature);

        if (deltaE > 0 || deltaET > Math.random()) {
            currentState = randomNeighbor;
            currentScore = neighborScore;

        }
        else {
            stuckCounter++;

        }

        cubeNumbers = [...currentState];
        renderNumbers();

        scores.push(currentScore);

        // Simpan hanya deltaET yang signifikan
        if (deltaET !== 1 && deltaET > 1e-10) {
            deltaETValues.push(deltaET);
        }

        if (stuckCounter >= stuckThreshold) {
            stuckIterations.push(iteration);
            stuckCounter = 0;
        }
        temperature *= coolingRate;

        if (temperature < 1e-10) {
            break;
        }
    }

    const simulatedannealingEndTime = Date.now();

```

```

        return {
            initialState: initialCubeArray,
            finalState: currentState,
            finalScore: currentScore,
            scores,
            duration: (simulatedannealingEndTime - simulatedannealingStartTime) /
1000,
            deltaETValues,
            stuckIterations : stuckCounter,
            iteration: iteration
        };
    }
}

```

- function generateRandomNeighbor(cubeArray)

Fungsi ini berfungsi untuk mencari neighbor secara acak. Fungsi ini menerima array yang berisi nilai-nilai dari magic cube. Kemudian, fungsi ini akan menukar dua nilai dalam magic cube berdasarkan indeks yang dipilih secara acak (menggunakan *Math.random()* pada panjang array untuk mendapatkan indeks secara acak). Setelah penukaran, fungsi akan mengembalikan array cube yang berisi dua nilai yang telah ditukar secara acak.

```

function generateRandomNeighbor(cubeArray) {
    const newNeighbor = [...cubeArray];
    const i = Math.floor(Math.random() * cubeArray.length);
    const j = Math.floor(Math.random() * cubeArray.length);

    [newNeighbor[i], newNeighbor[j]] = [newNeighbor[j], newNeighbor[i]];

    return newNeighbor;
}

```

## 4) Genetic Algorithm

Genetic Algorithm adalah salah satu jenis Algoritma local Search yang menggunakan persilangan (crossover) dan mutasi (mutation) dengan

harapan mendapat solusi yang lebih baik. Pada dasarnya alur pada algoritma ini adalah sebagai berikut.

- Inisialisasi populasi awal sebanyak 10 individu
- Loop dimulai dengan pengurutan array pada populasi awal berdasarkan fitness function
- Ambil dua individu dengan fitness function tertinggi
- Lakukan Crossover pada kedua individu tersebut
- Lakukan mutasi dengan menukar value pada child hasil crossover berdasarkan dengan mutation probability nya
- Masukkan child tersebut ke populasi awal
- Loop terus terjadi sampai kondisi terminasi tercapai.

Implementasi algoritma genetic algorithm digunakan untuk merealisasikan alur pencarian solusi diatas. Implementasi dilakukan dengan membuat berbagai function seperti dibawah ini.

- `InitPopulation(size)`

Function ini digunakan untuk inisialisasi populasi awal yang nantinya digunakan sebagai individu untuk crossover. Function ini menerima parameter size berupa jumlah individu yang ingin diinisialisasi. Individu - individu ini akan dibuat dengan value yang di generate secara random menggunakan `shuffleArray()`. Pada implementasi algoritma ini, jumlah individu yang diinisialisasi sebanyak 10 individu yang digenerate masing - masing dengan random. Berikut source code untuk function ini.

```
function initPopulation(size) {
    const population = [];
    for (let i = 0; i < size; i++) {
        const individual = [];
        for (let j = 1; j <= 125; j++) {
            individual.push(j);
        }
        shuffleArray(individual);
        population.push(individual);
    }
}
```

```
    return population;
}
```

- crossover(parent1, parent2)

Function ini digunakan untuk crossover dua individu dengan fitness function tertinggi untuk dijadikan parent. Function ini menerima parameter parent1 dan parent2 berupa 2 array individu dengan fitness function tertinggi. Function ini akan memilih titik crossover pada masing - masing parent secara random menggunakan math.random() kemudian dilakukan penggabungan value kedua parent tersebut berdasarkan titik potong tadi. Value yang sama pada kedua parent akan diisi dengan null pada salah satunya yang kemudian diisi angka dengan sesuai menggunakan function fillMissingNumbers(child). Berikut source code untuk function ini.

```
function crossover(parent1, parent2) {
  const child = Array(125).fill(null);
  const crossPoint = Math.floor(Math.random() * 125);
  const useNumber = [];

  for (let i = 0; i < crossPoint; i++) {
    child[i] = parent1[i];
    useNumber.push(parent1[i]);
  }
  for (let i = crossPoint; i < 125; i++) {
    let isUse = false;

    for (let j = 0; j < useNumber.length; j++) {
      if (useNumber[j] === parent2[i]) {
        isUse = true;
        break;
      }
    }
    if (!isUse) {
      child[i] = parent2[i];
      useNumber.push(parent2[i]);
    }
  }
}
```

```

    }
    return MissingNum(child);
}

```

- MissingNum(child)

Function ini digunakan untuk mengisi nilai null pada salah satu parent dengan value unik yang belum ter assign pada proses crossover. Function menerima parameter berupa array yang ingin diisi nilai null-nya. Function ini akan melakukan loop untuk menyimpan value yang telah ada pada child pada suatu array. Function ini kemudian menggunakan loop untuk mengisi nilai null pada child dengan value array yang berkisar 1 - 125 namun tidak terdapat pada array usedNumber[]. Berikut source code untuk function ini.

```

function MissingNum(child) {
    const numbers = [];
    for (let j = 1; j <= 125; j++) {
        numbers.push(j);
    }

    const numberUse = [];
    child.forEach(num => {
        if (num !== null) {
            numberUse.push(num);
        }
    });

    const filterNum = [];
    for (let i = 0; i < numbers.length; i++) {
        let isUse = false;
        for (let j = 0; j < numberUse.length; j++) {
            if (numbers[i] === numberUse[j]){
                isUse = true;
                break;
            }
        }
        if (!isUse) {
            filterNum.push(numbers[i]);
        }
    }
}

```

```

    }

    for (let i = 0; i < child.length; i++) {
        if (child[i] == null) {
            child[i] = filterNum.pop();
        }
    }
    return child;
}

```

- `mutate(individual)`

Function ini dilakukan untuk melakukan mutasi dan menghindari putaran solusi yang sama. Function menerima parameter berupa array yang ingin dilakukan mutasi. Fungsi ini menerima parameter berupa array individu. Kedua index pada array individu tersebut dipilih secara random dengan `math.random()`. Value dari kedua index yang terpilih kemudian ditukar. Berikut source code untuk function ini.

```

function mutate(individual) {
    const index1 = Math.floor(Math.random() * 125);
    const index2 = Math.floor(Math.random() * 125);
    let i = individual[index1];
    individual[index1] = individual[index2];
    individual[index2] = i;
}

```

- `geneticAlgorithm(maxPop, maxIteration)`

Function ini merupakan function utama untuk menjalankan algoritma Genetic dalam menemukan solusi untuk permasalahan magic cube. Function ini menerima dua parameter yaitu `maxPopulationSize` yang digunakan sebagai batasan ukuran populasi maksimum yang dapat dibentuk dan `maxIteration` sebagai batasan jumlah iterasi maksimum. Fungsi ini akan melakukan loop untuk menjalankan fungsi - fungsi lainnya seperti crossover dan

mutate sampai kondisi terminasi tercapai. Function ini akan return state awal, state akhir, durasi, skor final, jumlah iterasi, dan ukuran populasi . Berikut source code untuk function ini.

```
function geneticAlgorithm(maxPop, maxIteration) {
    let population = initPopulation(10);
    let startPop = population[0];
    let bestIndividual = population[0];
    let bestFitness = evaluate(bestIndividual);
    let generationCount = 0;
    let scores=[];
    const start = Date.now();

    while (generationCount < maxIteration && population.length <= maxPop) {
        generationCount++;
        population.sort(function(a, b) {
            return evaluate(b) - evaluate(a);
        });

        if (evaluate(population[0]) > bestFitness) {
            bestIndividual = population[0];
            bestFitness = evaluate(bestIndividual);
        }
        scores.push(bestFitness);

        const parent1 = population[0];
        const parent2 = population[1];
        let child = crossover(parent1, parent2);

        if (Math.random() < mutateProb) {
            mutate(child);
        }
        population.push(child);

        if (population.length >= maxPop || bestFitness == 109) {
            break;
        }

        console.log(`Generation ${generationCount}: Best Fitness =
${bestFitness}`);
    }
    const end = Date.now();
    const durations = (end - start) / 1000;
    console.log(`Algorithm stopped after ${generationCount} generations.`);
}
```

```

        return {
            initialState: startPop,
            finalState: bestIndividual,
            finalScore: bestFitness,
            scores,
            duration:durations,
            iteration:generationCount,
            lengthpop: population.length,
        };
    }
}

```

Kondisi terminasi untuk algoritma ini ada 3, yaitu ketika objective function mencapai nilai tertinggi (109), batas iterasi maksimum tercapai, dan batas ukuran populasi tercapai. Namun kondisi terminasi ketika objective function tertinggi tercapai memiliki peluang terjadi yang sangat kecil sehingga kondisi terminasi batas maksimum iterasi dan ukuran populasi yang akan terjadi.

## **C. Hasil eksperimen dan analisis (disertai dengan visualisasi dari program yang telah dibuat)**

Pada bagian ini akan dilakukan eksperimen penggunaan Algorithm sebagai solusi penyelesaian masalah magic cube. Kemudian, Hasil eksperimen akan dianalisis untuk feasibility, tingkat keoptimalan, perbandingan dengan algoritma lain serta aspek - aspek lainnya. Percobaan akan dilakukan sebanyak 3 kali dengan hasil eksperimen tersebut seperti.

- 1) State awal dan akhir dari kubus,
- 2) Nilai objective function akhir yang dicapai,
- 3) Plot nilai objective function terhadap banyak iterasi yang telah dilewati.
- 4) Durasi proses pencarian,
- 5) Banyak iterasi (Steepest Hill Climbing, Stochastic Hill Climbing, dan Genetic algorithm),
- 6) Plot  $e^{\frac{\Delta E}{T}}$  terhadap banyak iterasi yang telah dilewati (Simulated Annealing),

- 7) Frekuensi ‘stuck’ di local optimal (Simulated Annealing),
- 8) Jumlah populasi (Genetic algorithm).

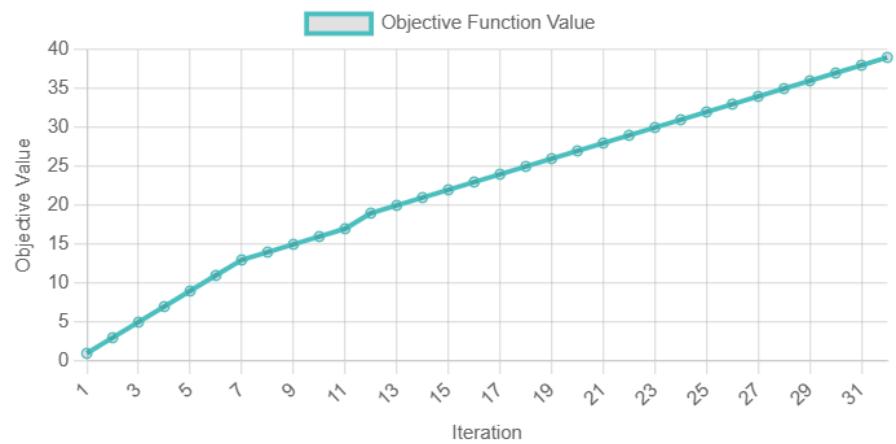
## 1) Steepest Ascent Hill Climbing

Berikut merupakan hasil percobaan dari algoritma Steepest Ascent Climbing:

1. Percobaan 1

**Final Score: 39**  
**Iterations: 32**  
**Duration: 0.718 seconds**

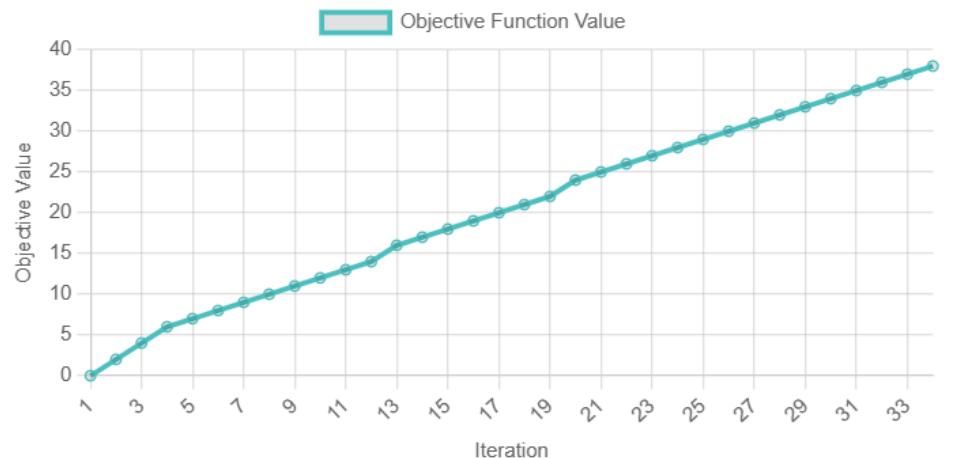
State Awal :									
37	40	38	74	95	31	117	94	80	98
50	119	19	14	35	8	55	124	15	28
18	65	58	26	70	17	7	60	32	49
91	34	30	51	111	47	102	12	20	56
82	25	23	6	100	103	108	2	53	68
125	109	90	63	61	121	83	54	13	78
75	85	88	22	105	27	84	71	72	113
93	4	52	39	116	44	89	11	9	10
62	112	36	77	66	118	43	1	33	73
106	5	64	59	104	86	76	29	24	92
State Akhir :									
37	40	20	111	95	69	117	94	80	114
50	119	97	14	35	93	55	124	15	28
96	65	58	26	70	17	12	60	87	49
91	66	38	46	74	33	23	7	24	56
82	25	102	6	100	103	108	30	41	68
122	69	57	48	3	122	2	57	48	44
45	120	16	21	114	45	120	16	36	98
101	42	81	46	115	101	84	18	31	115
87	110	67	96	123	51	110	67	121	123
107	97	41	79	99	107	19	11	79	99
104	109	90	63	61	81	34	54	13	78
75	85	88	113	105	27	42	116	72	62
8	4	52	39	71	3	89	53	9	10
22	112	21	77	83	118	43	1	47	73
106	5	64	59	125	86	76	29	32	92



## 2. Percobaan 2

**Final Score: 38**  
**Iterations: 34**  
**Duration: 0.608 seconds**

State Awal :									
16	51	98	121	58	26	64	114	40	34
24	41	123	1	25	20	102	112	103	36
96	108	74	115	2	18	117	37	105	87
43	125	53	23	65	95	63	106	19	92
49	69	8	100	30	79	110	94	89	59
State Akhir :									
77	11	122	47	78	44	21	113	60	77
9	70	81	50	90	62	23	107	75	48
5	32	13	86	104	72	22	92	52	76
10	17	82	93	42	120	12	28	4	35
84	55	91	54	118	109	111	3	15	56
47	69	20	121	58	19	108	114	40	34
8	41	123	118	25	98	102	5	15	95
16	64	74	115	46	18	117	37	105	39
43	29	53	125	65	101	63	106	66	88
49	112	24	100	30	79	71	94	89	59
7	11	122	26	78	44	21	113	60	77
9	70	96	50	90	62	23	107	75	48
51	32	13	12	104	72	22	92	52	76
81	17	82	93	42	120	12	28	4	35
84	55	91	54	1	109	111	3	124	56



### 3. Percobaan 3

Final Score: 36

Iterations: 29

Duration: 0.352 seconds

State Awal :

25	71	11	108	19
22	52	78	58	118
101	66	93	81	20
96	74	109	4	63
10	86	82	100	111

87	26	54	75	6
113	90	40	15	1
88	46	84	18	8
85	62	1	13	1
112	56	17	51	1

47	89	42	41	120
72	119	122	39	43
38	65	9	16	94
76	61	28	50	12
98	124	49	125	6

121	27	59	79	21
23	8	2	73	7
95	110	99	32	80
115	34	91	31	44
97	69	116	105	70

State Akhir :

25	71	92	108	19
83	52	40	22	118
101	38	93	81	2
96	74	78	4	63
10	86	12	100	111

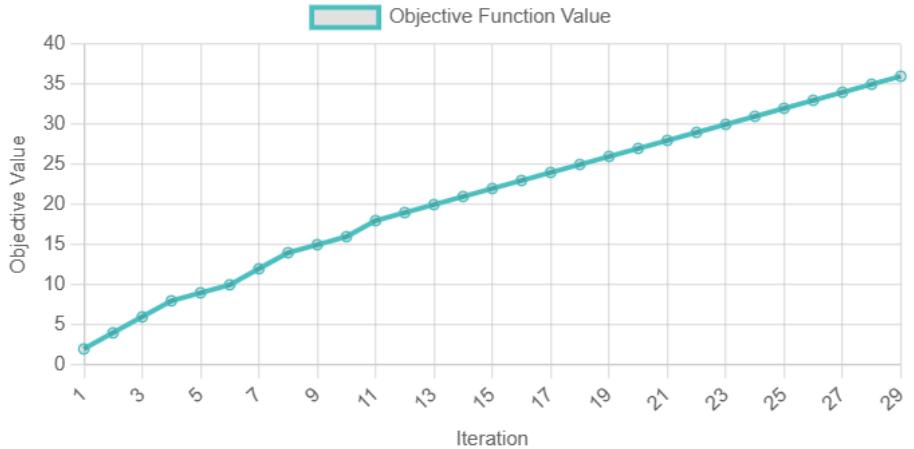
24	26	54	102	10
60	90	3	87	75
88	67	84	18	51
76	62	61	13	10
112	70	17	51	10

106	89	42	85	120
72	119	122	39	43
110	65	66	16	64
41	37	28	50	82
11	124	49	125	6

121	35	59	79	21
23	8	114	73	7
95	9	99	32	80
115	34	91	31	44
97	69	116	105	56

117 94 68 113 30

77	46	36	1	27
48	29	47	123	15
53	5	57	33	98
20	55	107	45	14



Dari hasil eksperimen tersebut, algoritma Steepest Ascent Hill Climbing memiliki kemampuan yang terbatas dalam mencapai global optimum. Algoritma ini secara konsisten mencapai local optimum, tetapi sering kali gagal mencapai solusi optimal karena tidak dapat melompati solusi sementara yang kurang baik. Hal ini menyebabkan pencarian berhenti saat tidak ada tetangga yang lebih baik, meskipun masih jauh dari kondisi Magic Cube yang ideal.

Jika dibandingkan dengan algoritma local search lainnya, Steepest Ascent Hill Climbing memiliki durasi pencarian yang relatif lebih cepat karena hanya mengevaluasi tetangga terbaik pada setiap iterasi, tanpa proses eksplorasi yang lebih acak atau intensif. Namun, pendekatan ini membuatnya rentan terjebak di local optimum, sehingga hasil akhirnya biasanya tidak seoptimal algoritma lainnya.

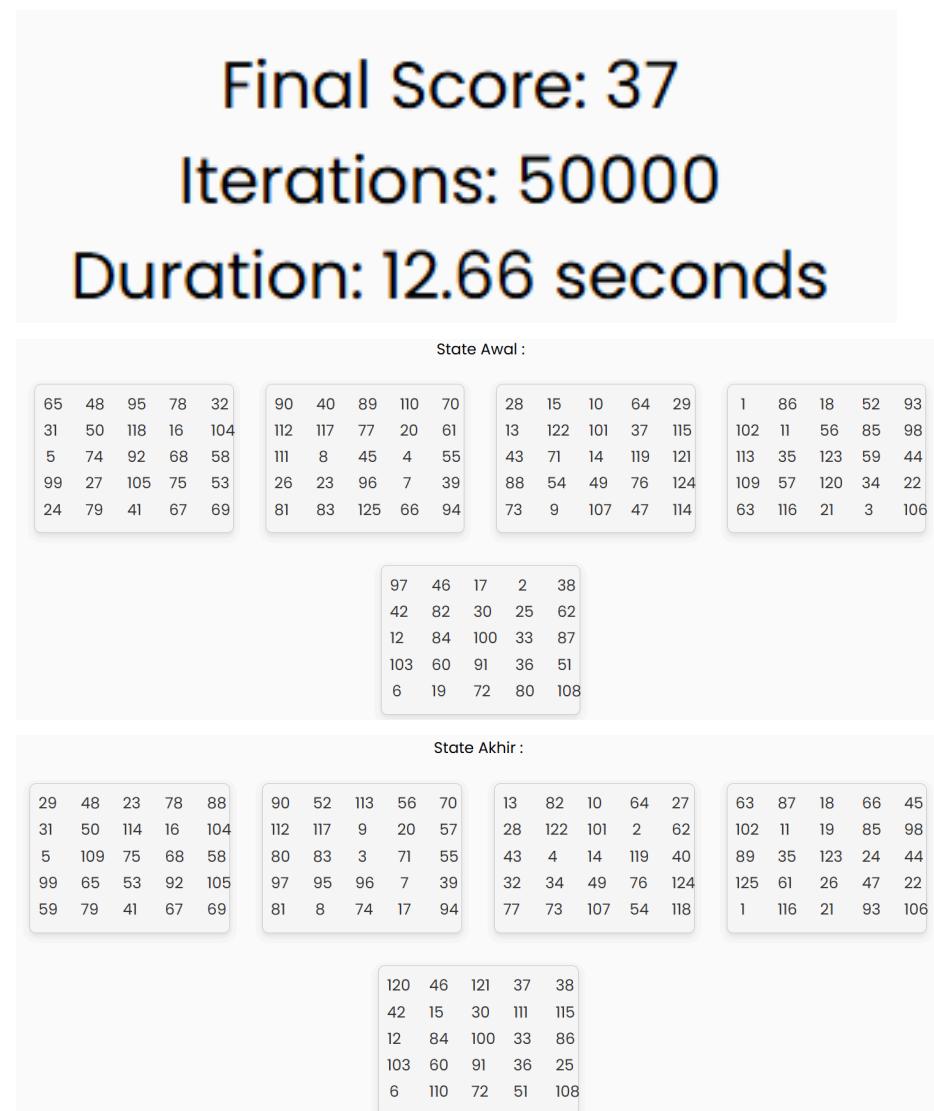
Dari segi konsistensi, algoritma ini memiliki hasil akhir yang cukup stabil antara percobaan yang satu dengan lainnya, meskipun pencapaian akhirnya tidak selalu maksimal. Dengan jumlah iterasi yang terbatas, algoritma Steepest Ascent Hill Climbing cenderung menghasilkan hasil yang mirip pada setiap

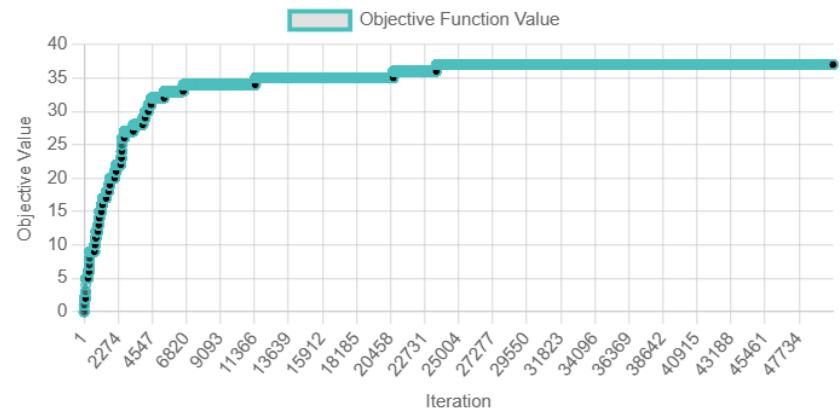
percobaan. namun, keterbatasan ini justru membatasi kemampuan untuk mencapai hasil yang lebih mendekati global optimum.

## 2) Stochastic Hill Climbing

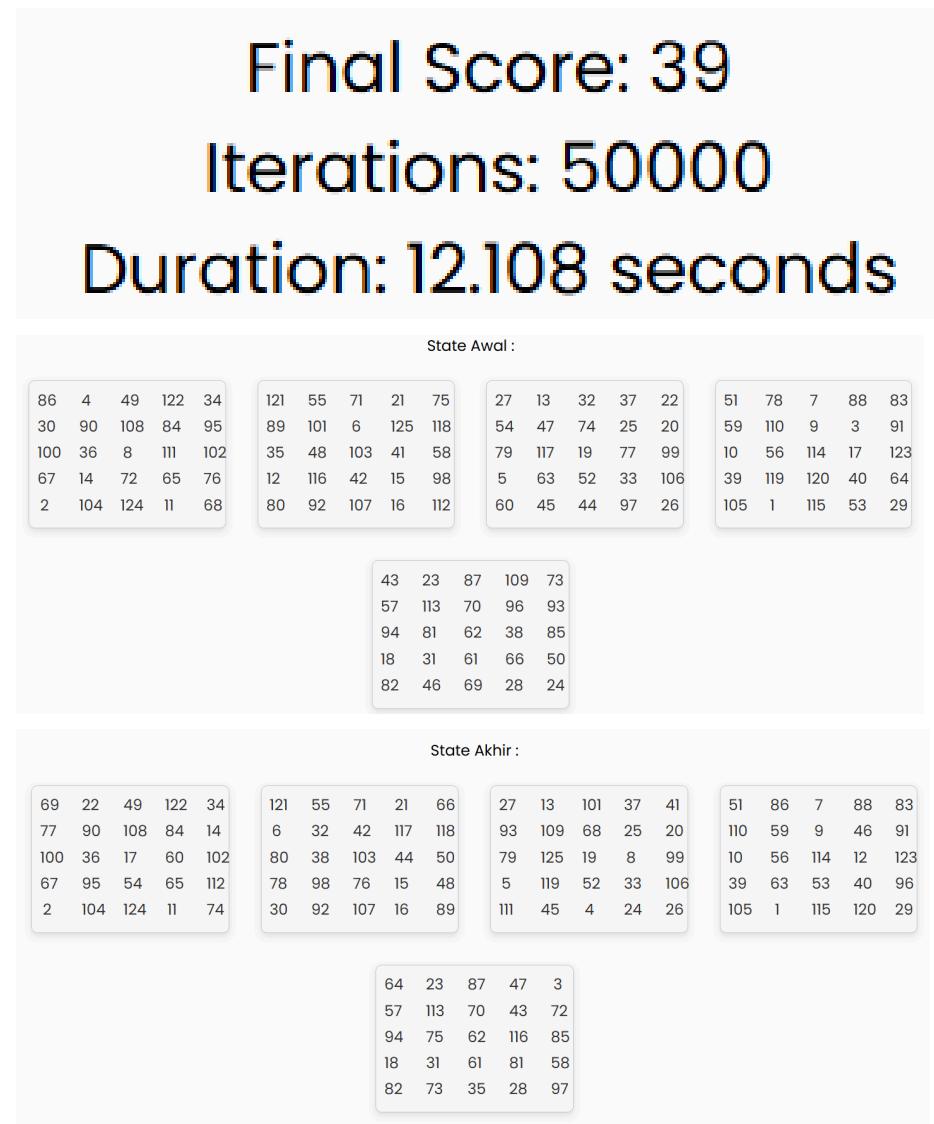
Berikut hasil percobaan dari stochastic hill climbing menggunakan ketentuan yang tertera di atas.

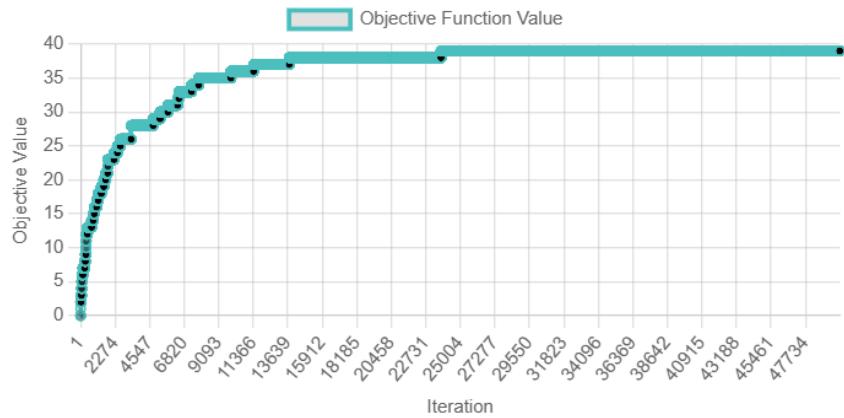
1. Percobaan 1



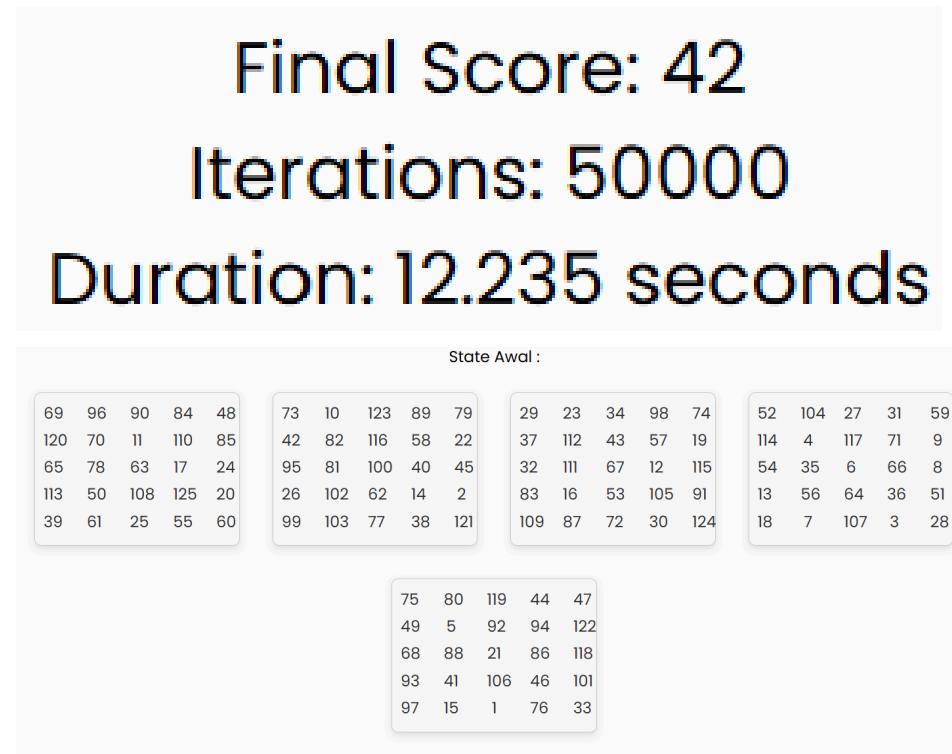


## 2. Percobaan 2

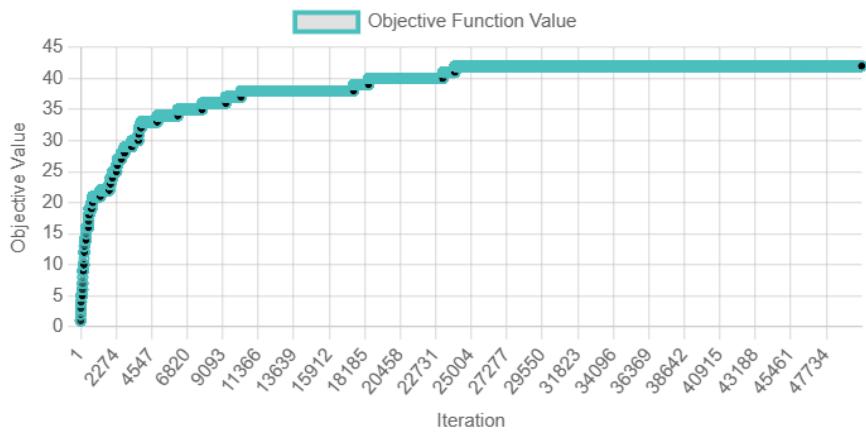




### 3. Percobaan 3



State Akhir :																
86	51	90	48	37	73	10	64	89	79	29	23					
120	22	11	77	85	42	82	34	58	80	84	112					
65	78	63	17	24	46	3	100	116	45	32	111					
113	47	26	70	59	55	102	7	14	101	83	69					
39	103	25	108	40	99	61	110	38	121	62	87					
52	54	8	36	124	114	4	117	71	9	104	35					
104	35	109	96	13	27	56	123	31	50	18	49					
27	56	123	31	50	18	49	107	81	119							
75	6	93	44	30	125	91	94	92	122	68	88					
68	88	21	20	118	28	41	106	95	16	97	15					
97	15	1	76	33												



Berdasarkan hasil dari 3 percobaan, hasil objective function yang didapatkan berada pada kisaran 37 - 42. Pada grafik terlihat bahwa terjadi stagnan pada objective function yang menyimpulkan bahwa meskipun algoritma stochastic hill climbing ini tidak akan berhenti selama belum mencapai max iterasi, nilai objective function yang dihasilkan tidak akan bertambah jika memang sudah tidak ada nilai neighbor yang lebih baik (stuck local optima).

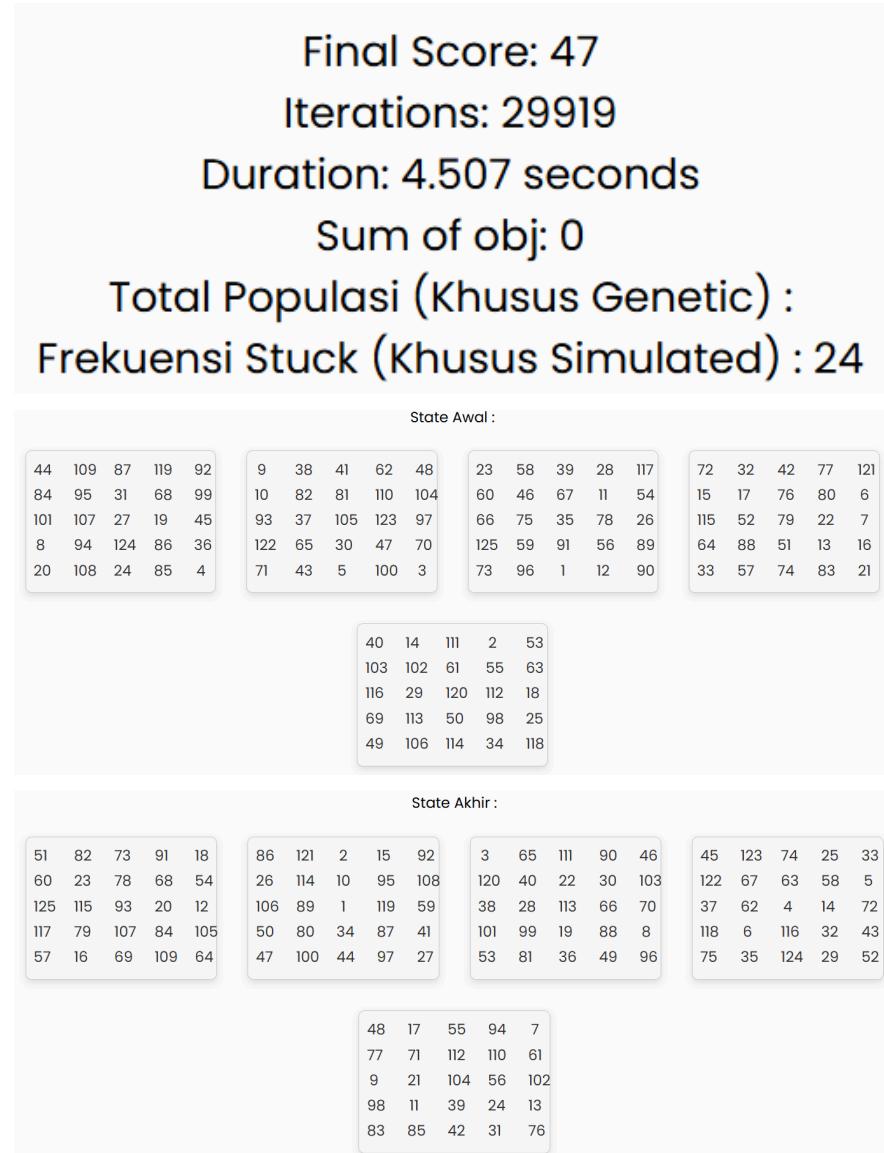
Hasil pencarian algoritma ini jika dibandingkan dengan yang lain tidak lebih baik jika dibandingkan dengan genetika dan simulated annealing karena masih tetap terjebak di local optima. Untuk durasi, jika dibandingkan dengan yang lain algoritma ini membutuhkan durasi yang lama. Semakin banyak iterasi akan menghasilkan objective function yang semakin baik, namun jika

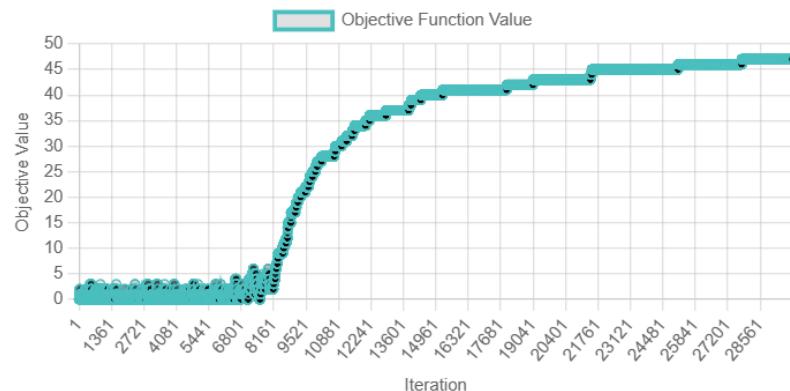
sudah tidak menemukan yang lebih baik maka objection function akan tetap meskipun terus melakukan iterasi.

### 3) Simulated Annealing

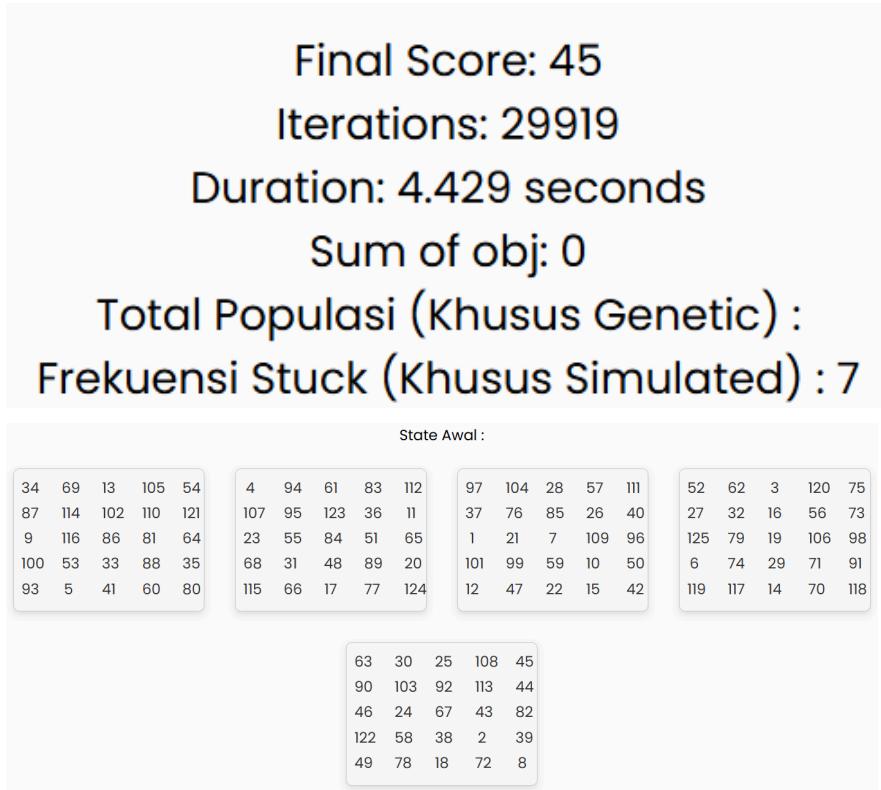
Berikut hasil:

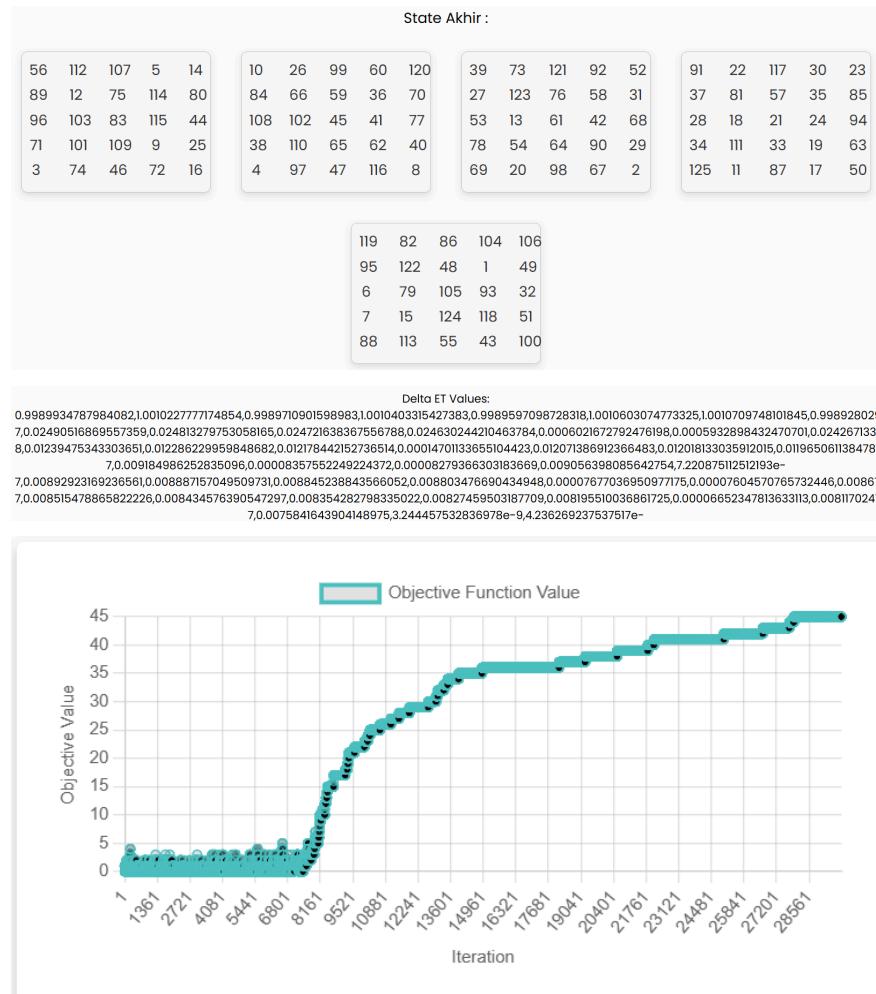
- Percobaan 1





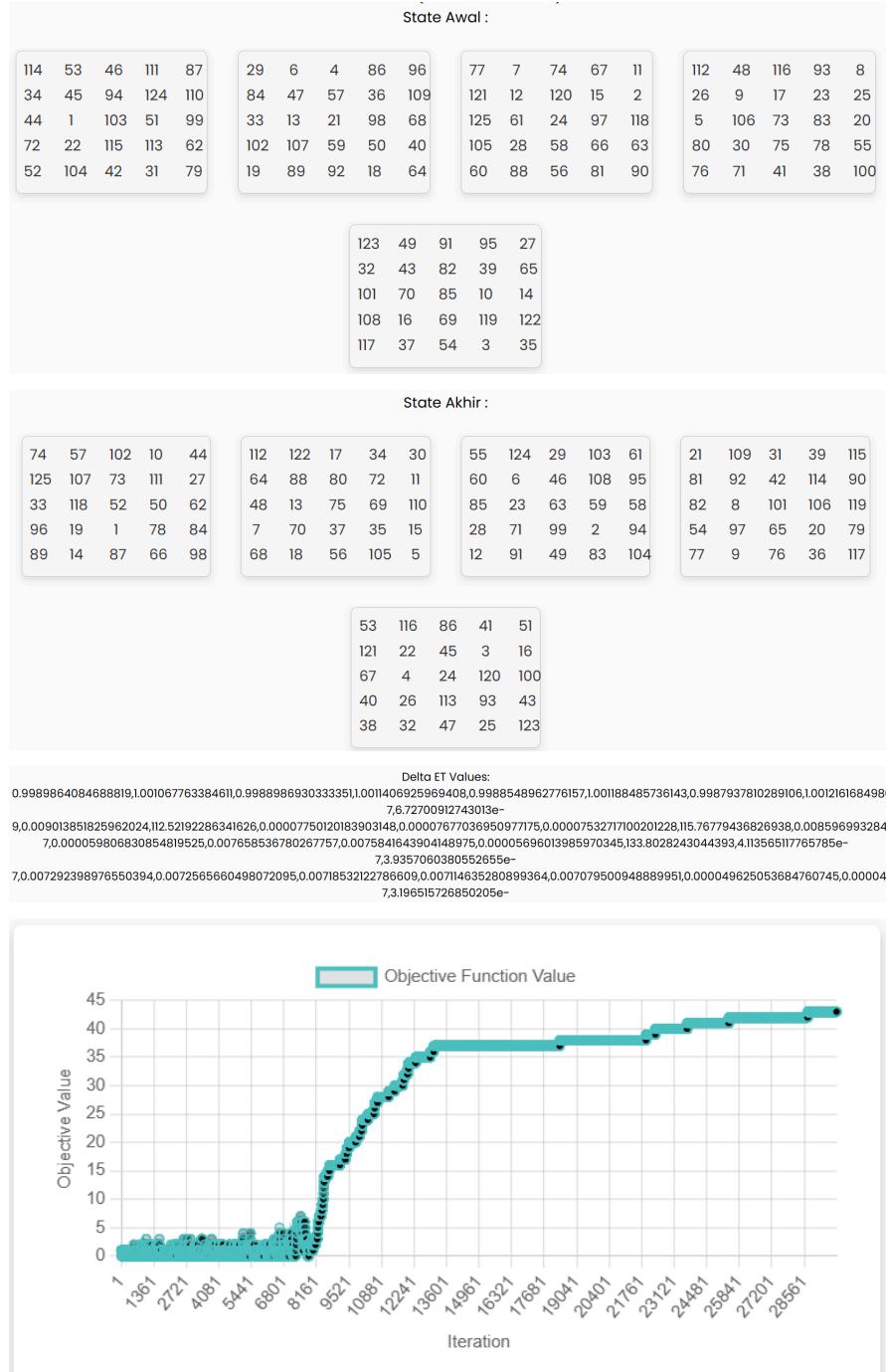
## 2. Percobaan 2





### 3. Percobaan 3

**Final Score: 43**  
**Iterations: 29919**  
**Duration: 5.28 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) :**  
**Frekuensi Stuck (Khusus Simulated) : 20**



Berdasarkan hasil percobaan dari berbagai algoritma, terlihat bahwa simulated annealing memiliki sejumlah keunggulan yang membuatnya menonjol dalam menghasilkan solusi yang optimal. Dalam tiga kali percobaan, algoritma ini konsisten mencapai nilai objective function di kisaran 43 hingga 47,

menunjukkan stabilitas yang cukup baik untuk mendekati solusi yang optimal. Simulated annealing memiliki mekanisme unik, yaitu penurunan temperatur bertahap yang dikombinasikan dengan pendekatan probabilistik. Hal ini memungkinkan algoritma untuk menghindari jebakan local optimum yang sering kali menghambat algoritma lainnya, seperti hill climbing.

Keunggulan utama dari simulated annealing adalah kemampuannya untuk “melompat” dari solusi suboptimal, mengurangi risiko terjebak pada local optimum. Dengan mekanisme ini, algoritma tidak hanya melakukan eksplorasi ruang solusi secara lebih efektif, tetapi juga menjaga stabilitas durasi pencarian di setiap percobaan. Artinya, algoritma ini mencapai keseimbangan antara eksplorasi dan eksploitasi yang memastikan pencarian solusi baru berjalan dengan lancar tanpa mengorbankan waktu komputasi.

Dibandingkan dengan algoritma lainnya, simulated annealing tidak memiliki kompleksitas pengaturan populasi seperti algoritma genetika, namun justru di sinilah keunggulan praktisnya. Tanpa perlu mengelola berbagai individu dalam sebuah populasi, algoritma ini tetap efisien secara komputasi dan sederhana dalam implementasinya, sambil mempertahankan kemampuan menemukan solusi yang mendekati optimal. Kesederhanaan ini membuatnya sangat ideal bagi kasus optimasi yang memerlukan keseimbangan antara waktu pencarian yang cepat dan hasil yang mendekati optimal.

Jika dibandingkan, algoritma lain mungkin menunjukkan keunggulan dalam konteks tertentu, namun simulated annealing memberikan keseimbangan yang unggul antara kesederhanaan,

stabilitas pencarian, dan hasil optimal. Keunggulan mekanisme probabilistiknya menjadi nilai tambah signifikan, menjadikannya pilihan yang sangat efisien untuk kebutuhan pencarian solusi optimal di berbagai situasi.

#### 4) Genetic Algorithm

- Variabel kontrol : maxIteration dengan maxIteration = 5000
  1. Percobaan 1

**Final Score: 15**  
**Iterations: 5000**  
**Duration: 18.41 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 5010**

State Awal :									
51	57	42	54	11	8	113	25	80	73
58	118	38	95	75	68	86	77	14	82
22	2	23	56	16	101	21	37	69	98
91	121	66	76	79	33	102	9	125	27
18	30	40	48	3	107	115	99	17	110
123	59	53	10	43	88	1	74	19	35
12	124	111	28	122	93	112	60	100	104
61	15	108	46	84					
44	90	4	103	96	85	63	39	94	45
97	6	92	13	106	47	109	71	52	119
31	67	70	50	120	7	64	32	34	26
36	5	89	117	83	65	78	116	81	62
114	49	41	87	24	29	55	105	20	72

State Akhir :

51	57	95	54	11
58	118	38	42	59
22	2	23	55	16
91	121	66	76	39
34	32	53	88	77

123	31	48	41	62
107	100	120	49	47
81	97	96	112	61
101	37	9	78	85
71	70	20	43	60

52	67	82	124	50
116	86	65	44	113
122	40	83	26	36
10	103	28	27	25
75	19	24	94	21

104	99	6	114	111
45	64	3	33	46
110	102	108	4	17
7	90	125	5	93
8	15	73	115	80

13	18	84	106	98
92	109	89	117	119
79	74	69	68	14
35	30	87	63	12
29	56	105	1	72



## 2. Percobaan 2

**Final Score: 12**

**Iterations: 5000**

**Duration: 18.03 seconds**

**Sum of obj: 0**

**Total Populasi (Khusus Genetic) : 5010**

State Awal :

122	86	80	11	19
40	112	108	41	94
16	1	65	63	105
29	96	60	121	47
34	109	103	115	67

7	72	85	83	50
66	90	92	64	13
52	45	62	98	44
79	116	26	51	55
97	111	54	58	15

38	91	75	20	49
46	14	57	87	106
5	56	71	82	69
27	114	30	9	70
17	43	102	119	99

6	32	21	89	42
88	12	107	68	28
117	84	33	53	24
125	74	10	61	35
3	77	22	120	101

93	18	37	4	113
118	110	100	123	81
39	104	73	76	95
2	59	48	25	8
78	31	23	124	36

State Akhir :

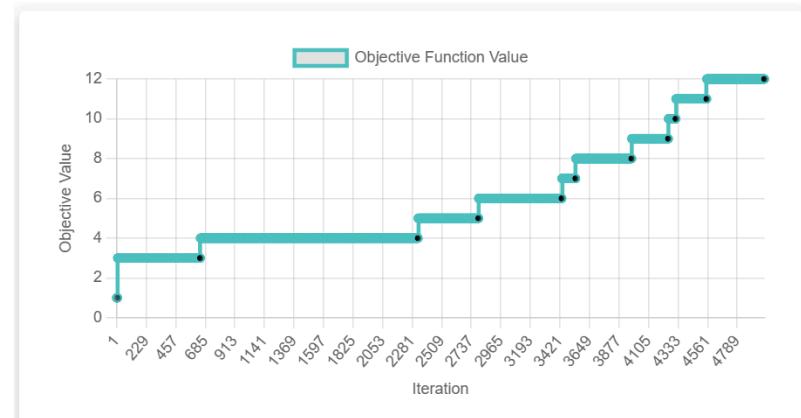
80	69	121	84	37
48	43	112	62	58
81	90	66	61	99
101	52	72	96	88
98	21	87	113	110

30	119	125	67	78
51	33	115	53	63
100	103	86	16	76
14	17	92	93	57
68	24	35	91	25

73	50	22	8	29
45	46	10	124	47
18	7	108	85	97
114	1	107	31	3
65	56	105	94	109

95	71	34	120	27
41	64	11	70	19
2	13	89	42	77
55	123	106	44	5
54	60	116	39	12

118	79	4	36	9
23	122	26	75	102
104	82	20	111	40
38	15	28	6	59
32	83	117	74	49



### 3. Percobaan 3

**Final Score: 14**  
**Iterations: 5000**  
**Duration: 18.039 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 5010**

State Awal :									
17	43	7	39	26	94	58	23	67	56
97	103	6	96	70	87	102	125	88	12
38	117	75	18	77	45	21	109	84	79
50	53	81	30	29	68	47	62	54	90
100	37	110	66	71	10	42	32	15	1
86	40	78	122	121	14	98	13	11	8
113	85	69	60	95	9	28	65	2	74
63	92	41	51	119	44	4	101	111	22
115	73	76	31	33	91	108	52	99	83
49	124	20	106	48	35	118	82	27	34

State Akhir :

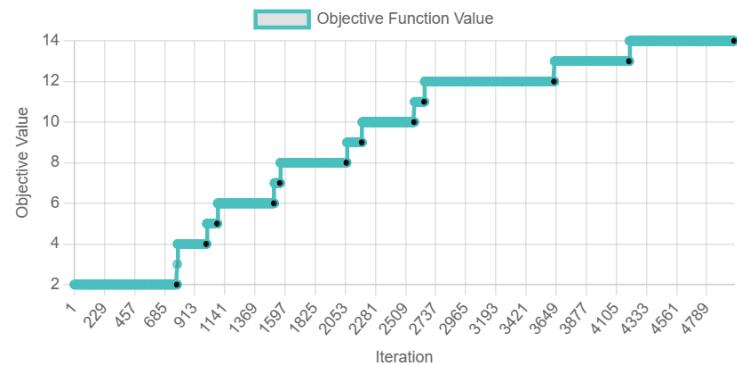
99	6	51	118	58
52	117	121	15	10
56	41	26	95	97
88	60	83	7	42
125	113	114	80	86

8	50	119	47	70
40	11	93	69	102
9	13	116	53	98
81	59	30	57	28
66	39	106	89	22

120	32	82	3	62
107	48	49	17	45
65	36	2	31	76
105	110	112	87	100
124	61	68	21	72

18	29	37	103	90
25	55	35	109	74
71	92	108	27	12
78	77	14	75	24
123	101	122	5	115

111	104	44	91	85
96	84	16	73	94
67	64	63	54	46
43	79	38	34	4
33	1	20	19	23



- Variabel kontrol : maxIteration dengan maxIteration = 7000

1. Percobaan 1

**Final Score: 15**  
**Iterations: 7000**  
**Duration: 35.419 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 7010**

State Awal :

11	107	88	8	68
40	97	45	23	116
93	60	18	9	102
90	24	83	72	79
15	114	70	119	30

118	103	3	63	91
33	52	44	122	2
37	112	19	82	113
5	53	92	95	48
120	17	104	10	124

99	74	76	59	32
46	43	27	12	101
21	61	25	55	39
50	65	4	34	105
89	106	28	84	31

66	26	29	20	14
58	1	86	47	123
36	62	75	111	16
100	22	108	6	42
110	13	80	49	54

57	121	77	98	85
7	87	117	38	109
41	78	73	35	67
81	94	115	64	51
125	69	96	56	71

State Akhir :

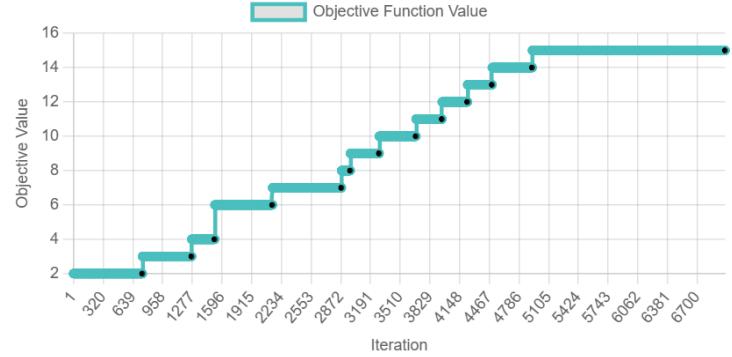
29	89	15	42	12
102	45	87	111	34
30	41	68	71	120
93	63	48	110	1
61	20	47	28	36

123	72	90	115	96
94	67	60	35	5
69	118	114	103	51
84	79	59	116	56
14	86	19	38	107

104	21	105	75	10
33	64	66	98	54
99	18	83	58	113
25	46	65	73	106
26	8	108	23	112

117	50	32	70	24
119	78	81	77	55
39	13	88	40	121
11	52	91	124	80
31	122	82	4	76

2	7	85	17	101
16	74	53	62	3
92	125	49	43	100
109	9	6	37	27
22	44	57	97	95



## 2. Percobaan 2

**Final Score: 18**  
**Iterations: 7000**  
**Duration: 35.624 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 7010**

State Awal :

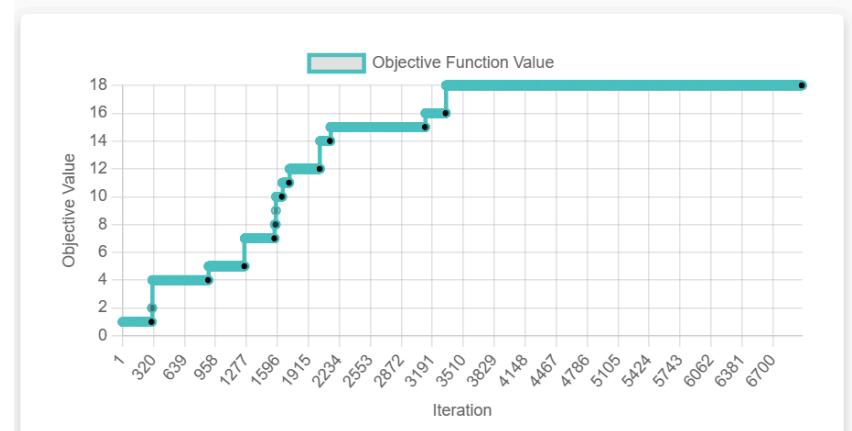
79	13	102	21	52	15	26	20	3	49	14	104	88	101	19
108	81	70	93	116	1	9	84	45	32	8	59	35	38	43
57	16	56	72	42	22	28	10	46	87	76	53	64	96	100
115	78	82	54	65	95	27	4	123	71	114	92	34	111	98
47	50	58	37	24	5	33	30	25	97	63	68	119	122	7

124	112	103	109	83	86	125	89	6	23
117	69	121	67	77	36	106	60	99	94
48	73	110	91	61	118	11	85	12	40
62	120	51	31	18	66	80	39	90	44
105	41	75	29	17	55	74	2	107	113

State Akhir :

79	13	102	100	21	31	8	71	113	92	123	38	3	40	62
84	36	95	68	82	34	55	58	18	104	59	99	76	89	70
72	78	46	109	10	42	86	33	17	37	29	117	124	20	26
103	9	51	107	30	125	66	32	54	116	63	44	27	111	43
97	14	108	94	47	83	106	81	19	61	41	6	122	93	114

35	25	22	4	101	12	74	64	48	39
52	23	110	90	87	2	73	50	80	88
60	85	49	56	118	112	75	5	119	121
91	120	16	67	96	115	69	65	57	53
77	28	105	98	7	45	24	15	11	1



### 3. Percobaan 3

Final Score: 20

Iterations: 7000

Duration: 35.991 seconds

Sum of obj: 0

Total Populasi (Khusus Genetic) : 7010

State Awal :

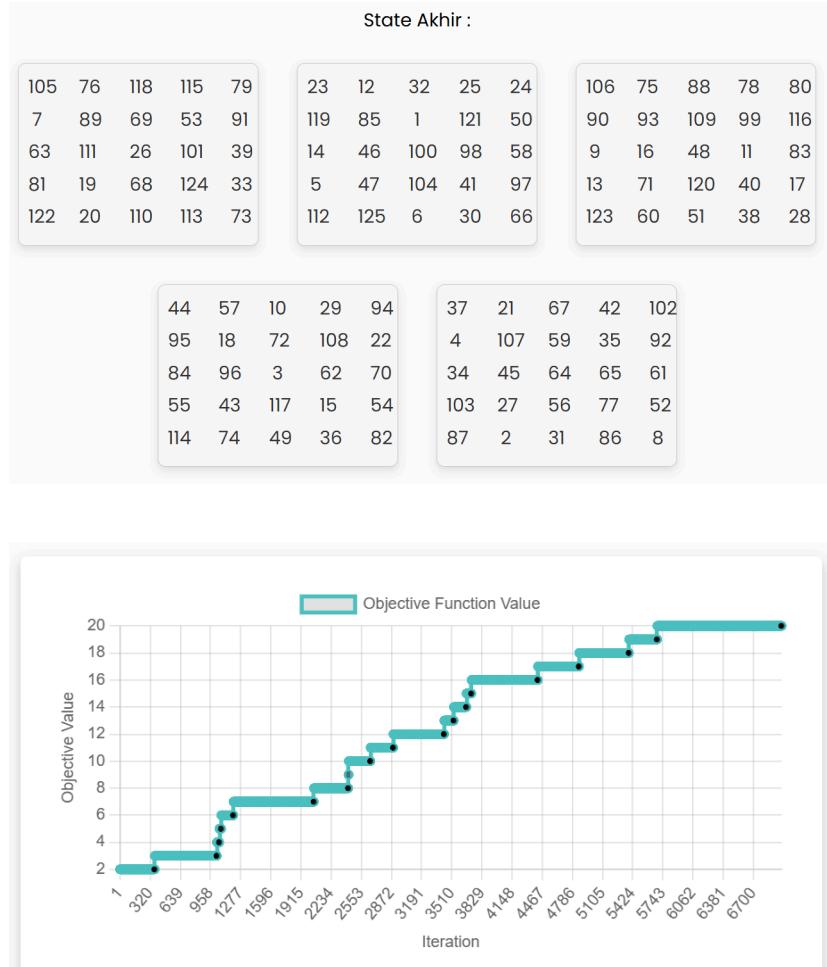
95	39	2	27	47
6	14	82	105	84
104	1	98	48	77
116	75	107	80	53
20	37	19	65	18

57	52	86	31	102
81	108	40	33	71
97	68	50	66	8
109	113	64	34	114
28	85	92	63	49

117	73	42	115	23
93	3	88	56	124
24	100	41	29	90
61	112	120	44	87
13	4	45	111	69

60	70	121	11	25
36	21	110	59	5
15	118	123	51	32
96	67	119	17	78
89	103	22	10	76

46	99	62	7	9
35	125	38	12	94
91	55	72	74	54
122	26	79	43	101
106	58	83	16	30



- Variabel kontrol : maxIteration dengan maxIteration = 10000
  1. Percobaan 1

**Final Score: 19**  
**Iterations: 10000**  
**Duration: 72.208 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 10010**

State Awal :

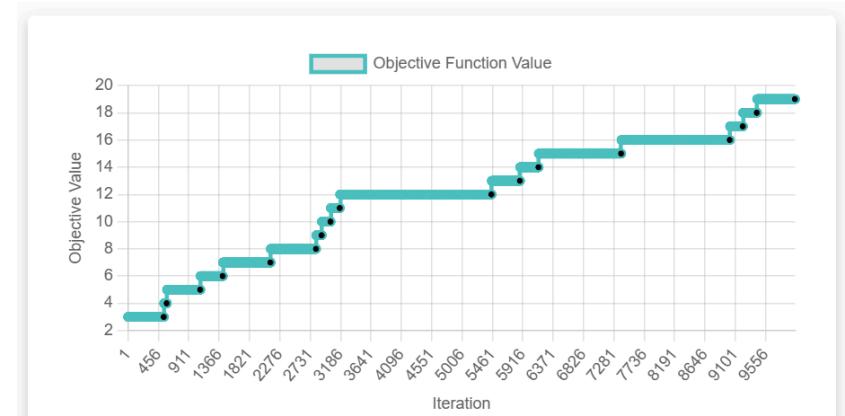
100	2	77	66	83	89	48	84	108	93	115	109	36	122	9
4	96	19	44	56	46	90	95	58	82	29	27	103	118	117
86	49	13	42	52	28	113	38	43	12	60	68	16	24	67
124	55	116	87	91	70	72	6	33	120	54	85	62	71	80
40	30	10	111	63	123	88	25	45	17	11	76	79	53	14

15	59	78	101	102	94	3	114	34	74
23	81	119	125	22	39	112	61	35	41
37	47	98	57	7	26	18	65	75	51
92	32	104	64	107	5	105	31	73	99
50	121	69	106	1	97	21	8	110	20

State Akhir :

2	77	7	9	103	107	116	106	118	17	82	25	19	62	94
122	124	28	58	59	65	1	47	21	89	114	39	110	33	37
74	85	92	93	104	113	29	38	91	44	99	112	41	15	10
67	64	13	76	23	20	100	81	63	102	52	105	4	31	53
50	66	86	117	83	95	69	11	79	61	42	8	6	70	98

45	120	75	24	51	34	3	73	5	78
49	125	46	115	97	68	26	84	88	119
80	54	96	72	30	71	35	48	90	36
32	43	111	123	121	55	14	56	22	101
18	60	27	12	16	87	108	109	40	57



## 2. Percobaan 2

**Final Score: 18**

**Iterations: 10000**

**Duration: 130.023 seconds**

**Sum of obj: 0**

**Total Populasi (Khusus Genetic) : 10010**

**State Awal :**

42	13	14	2	46
36	55	88	108	60
49	121	26	93	102
59	20	105	66	15
1	98	68	72	16

114	111	50	84	97
94	28	110	83	123
81	24	45	107	117
12	100	95	124	101
35	11	25	41	53

91	51	70	48	52
31	10	18	29	106
73	5	44	92	56
125	61	4	9	79
3	87	96	75	40

113	115	34	77	122
118	69	33	30	37
58	116	67	39	103
47	90	74	27	89
71	80	62	21	7

120	19	112	8	104
109	85	64	65	99
17	23	6	57	54
119	86	22	78	43
32	63	82	76	38

**State Akhir :**

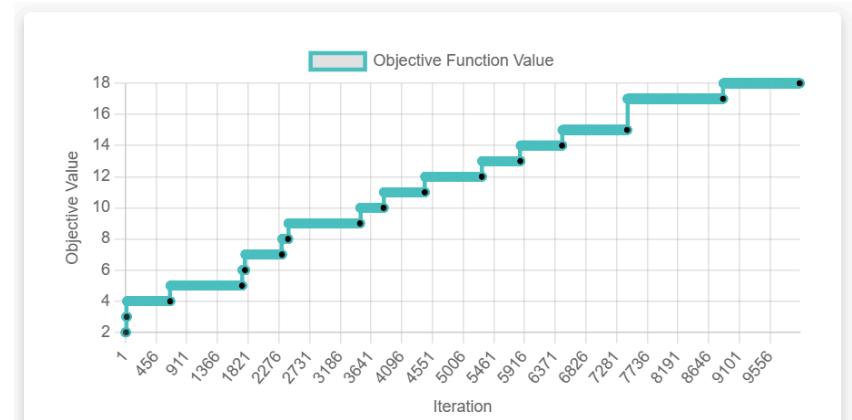
12	81	27	76	119
55	80	72	68	40
111	37	73	8	21
102	50	85	38	87
35	53	6	125	123

1	95	121	64	89
59	101	92	122	39
57	2	58	86	51
47	82	105	45	22
94	19	104	75	114

66	93	84	33	74
98	20	44	11	108
16	3	115	25	18
14	67	17	60	109
48	91	15	7	100

71	107	120	24	96
97	26	28	52	112
110	78	56	43	5
116	34	23	83	10
77	70	49	113	99

118	117	124	106	103
90	88	79	65	61
46	42	69	41	54
36	62	30	29	31
32	63	13	9	4



### 3. Percobaan 3

**Final Score: 20**  
**Iterations: 10000**  
**Duration: 111.053 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 10010**

State Awal :									
110	125	60	21	102	30	64	44	9	121
12	39	82	98	100	25	34	38	78	74
45	26	95	14	8	4	35	76	115	49
46	11	36	117	118	10	31	47	108	24
119	120	23	79	61	32	37	1	87	52
77	86	91	16	69	75	99	29	90	85
92	41	57	83	73	71	116	54	107	67
27	97	94	33	66	104	72	103	55	88
43	111	5	22	106	50	20	40	84	101
7	93	28	105	122	13	6	96	124	53

State Akhir :

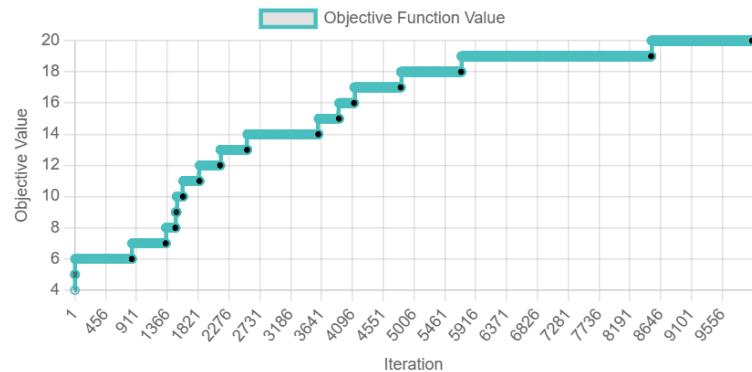
118	84	44	32	37
35	110	20	86	30
85	9	45	95	28
2	88	6	94	83
52	24	40	8	22

14	91	21	101	105
82	102	92	1	103
36	68	43	50	80
99	42	29	62	87
56	12	69	27	57

119	16	54	79	47
15	81	55	114	77
120	60	7	3	96
19	41	109	23	46
67	117	125	111	49

97	18	122	51	11
70	116	64	61	4
65	17	26	112	53
93	98	13	73	38
106	72	123	121	115

108	89	74	104	100
113	31	10	25	90
78	59	107	75	58
76	39	71	63	66
34	33	124	48	5



- Variabel kontrol : maxPop dengan maxPop = 5000

1. Percobaan 1

**Final Score: 13**  
**Iterations: 4990**  
**Duration: 20.152 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 5000**

State Awal :

91	41	104	108	13
122	35	45	117	36
109	18	124	9	65
94	92	23	93	89
99	34	79	67	52

75	81	74	88	25
8	51	15	114	101
105	106	73	125	87
78	28	12	20	48
68	31	46	84	33

6	19	54	56	121
90	113	70	86	100
115	71	98	77	96
69	62	1	76	24
97	37	116	10	11

29	43	58	60	123
53	103	44	107	3
59	39	118	16	85
82	63	21	83	55
2	95	14	7	110

State Akhir :

104	37	110	95	51
67	92	56	4	48
106	90	15	63	49
53	68	94	25	75
29	115	120	43	79

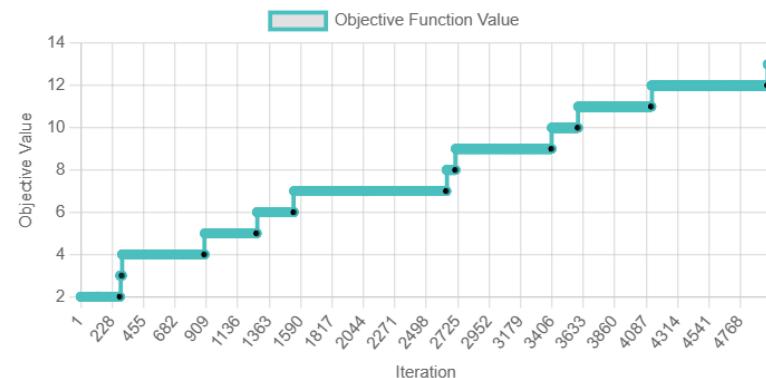
44	47	118	12	31
91	111	2	32	26
35	8	40	96	59
71	6	28	114	46
74	22	64	61	23

101	81	116	80	33
34	105	77	36	98
3	87	11	17	16
13	42	7	9	108
10	39	18	50	89

72	99	112	19	119
100	41	1	24	60
70	125	122	45	124
58	54	66	85	76
88	84	20	102	21

5	78	103	109	57
117	52	14	69	83
113	82	27	30	123
121	38	65	62	107
73	55	93	86	97



## 2. Percobaan 2

**Final Score: 11**  
**Iterations: 4990**  
**Duration: 20.329 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 5000**

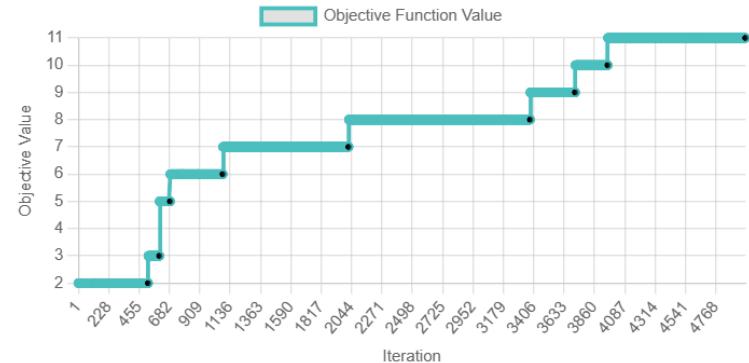
State Awal :									
121	74	64	7	94	117	9	5	83	114
24	113	66	32	70	10	93	115	17	49
124	27	88	80	23	125	79	76	72	1
61	101	87	22	90	36	18	26	60	43
92	8	100	89	123	25	105	28	51	111

State Akhir :									
51	73	19	122	101	30	119	25	59	124
6	62	46	91	55	18	80	75	57	85
22	108	92	16	56	123	114	36	27	31
106	48	45	34	109	8	102	111	112	77
32	24	3	49	116	84	67	82	15	47

Final Score: 12									
Iterations: 4990									
Duration: 20.478 seconds									
Sum of obj: 0									
Total Populasi (Khusus Genetic) : 5000									



### 3. Percobaan 3

**Final Score: 12**  
**Iterations: 4990**  
**Duration: 20.478 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 5000**

State Awal :

86	102	62	5	97
20	37	28	95	96
113	59	65	89	110
48	99	11	90	14
45	42	36	122	12

41	56	71	120	109
78	75	80	61	104
121	76	79	44	24
7	63	72	94	82
46	123	19	66	22

67	115	74	108	85
93	17	98	3	10
30	111	29	38	87
31	117	70	27	77
51	91	124	1	33

101	69	39	105	60
83	100	40	116	4
8	35	55	125	112
32	50	6	26	9
49	21	52	88	47

State Akhir :

51	90	60	82	41
88	59	101	91	53
96	65	27	97	30
71	112	114	117	4
9	105	13	67	95

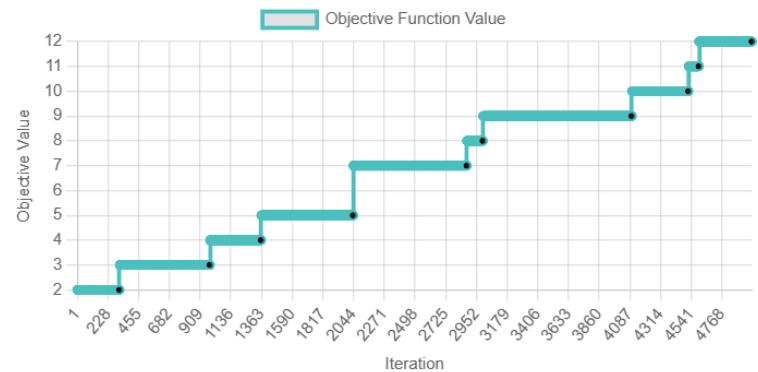
100	7	123	118	45
22	2	17	89	98
55	47	44	120	21
8	35	11	85	75
102	113	121	92	48

49	3	109	38	116
66	52	86	34	12
122	80	104	33	16
99	81	39	74	50
79	24	108	56	36

58	125	6	20	19
107	54	32	110	15
119	77	78	57	37
64	14	28	70	42
83	25	40	84	5

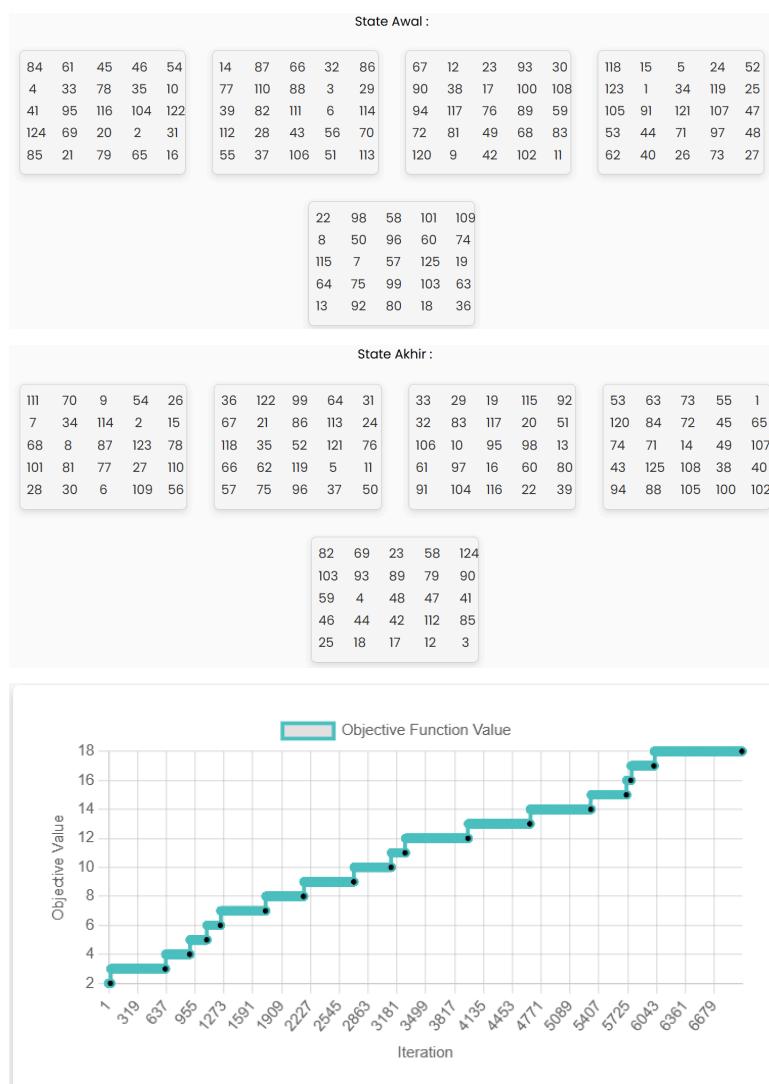
  

69	10	29	18	94
26	63	62	76	61
124	115	46	103	93
87	73	72	68	43
31	23	106	111	1



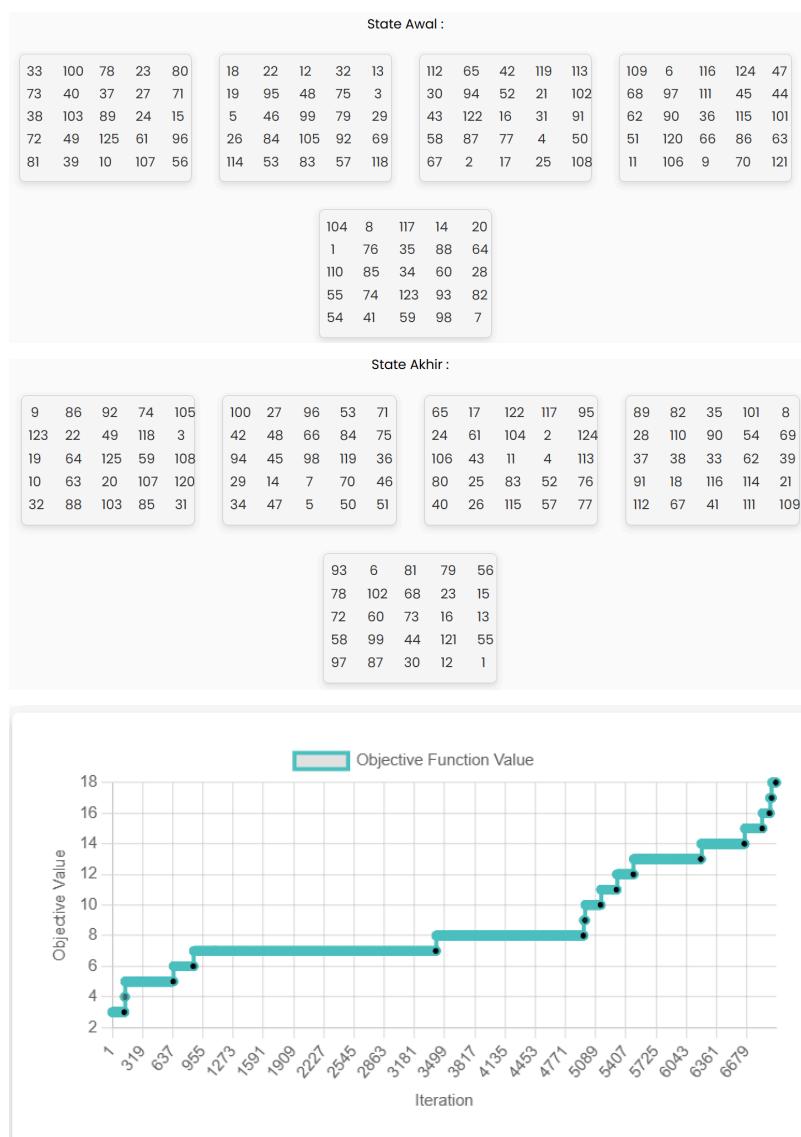
- Variabel kontrol : maxPop dengan maxPop = 7000
  1. Percobaan 1

**Final Score: 18**  
**Iterations: 6990**  
**Duration: 25.727 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 7000**



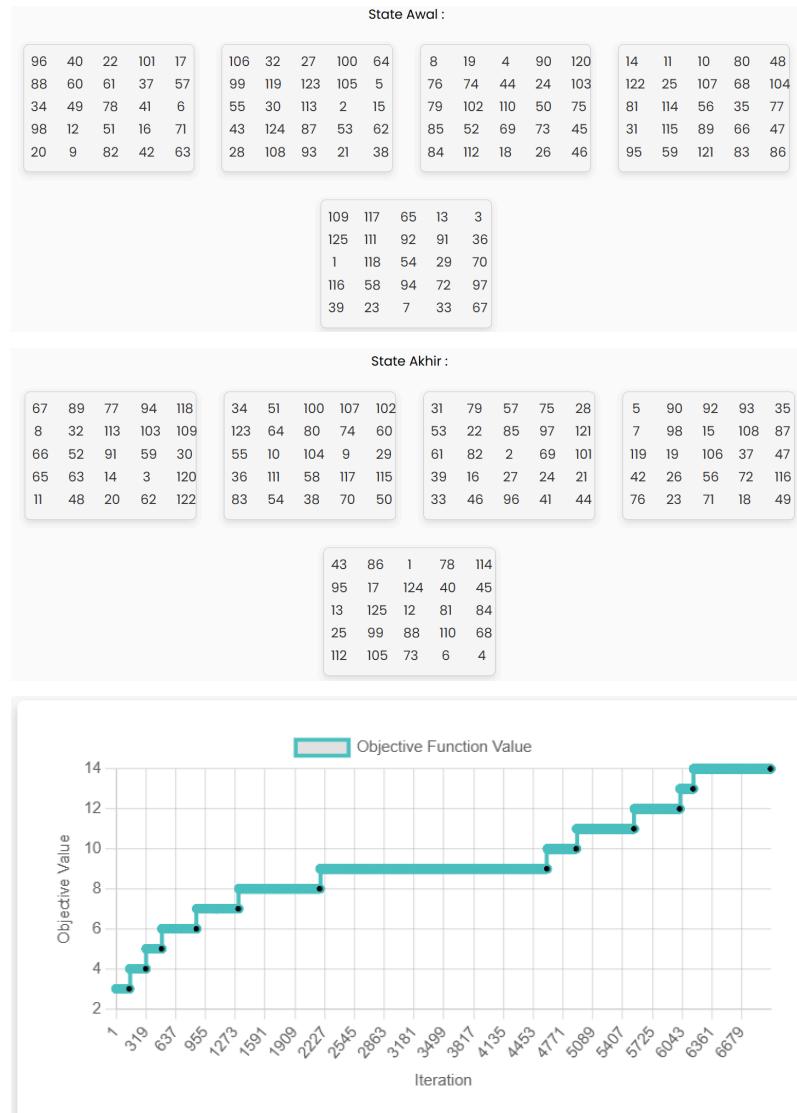
## 2. Percobaan 2

**Final Score:** 18  
**Iterations:** 6990  
**Duration:** 27.548 seconds  
**Sum of obj:** 0  
**Total Populasi (Khusus Genetic) :** 7000



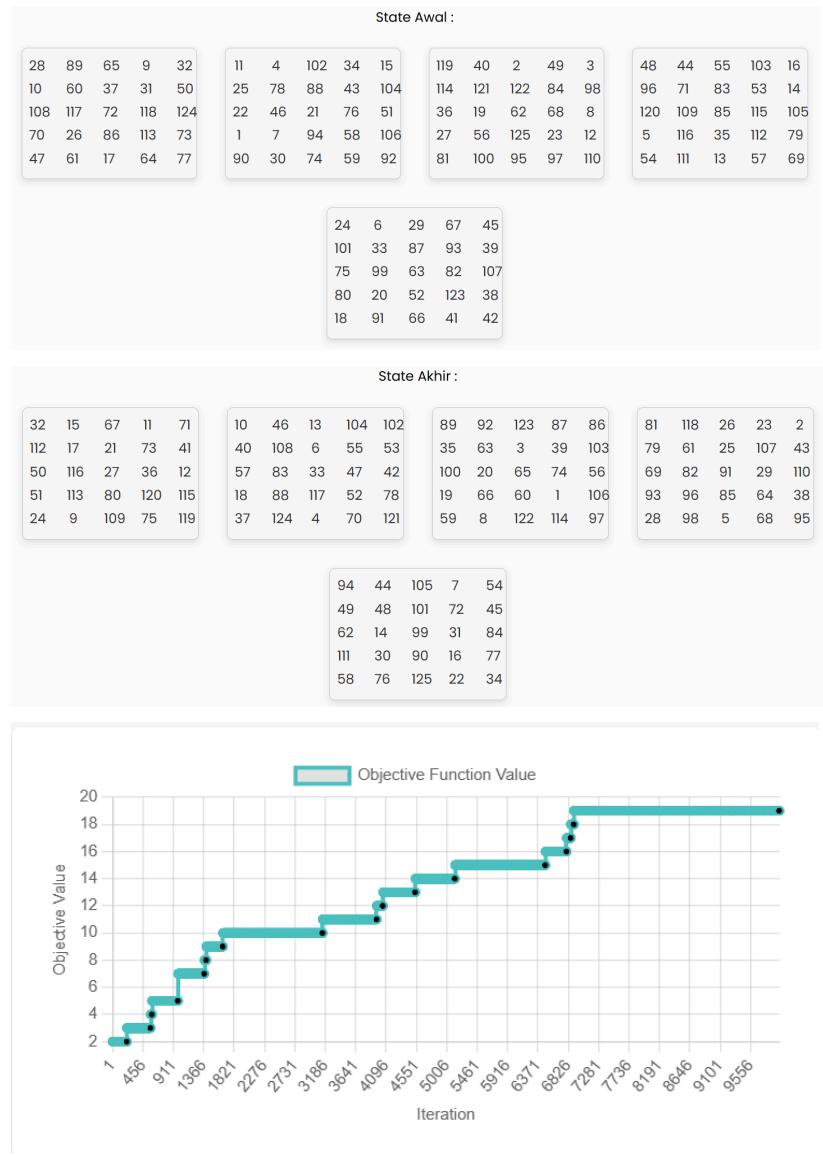
### 3. Percobaan 3

**Final Score: 14**  
**Iterations: 6990**  
**Duration: 25.993 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 7000**



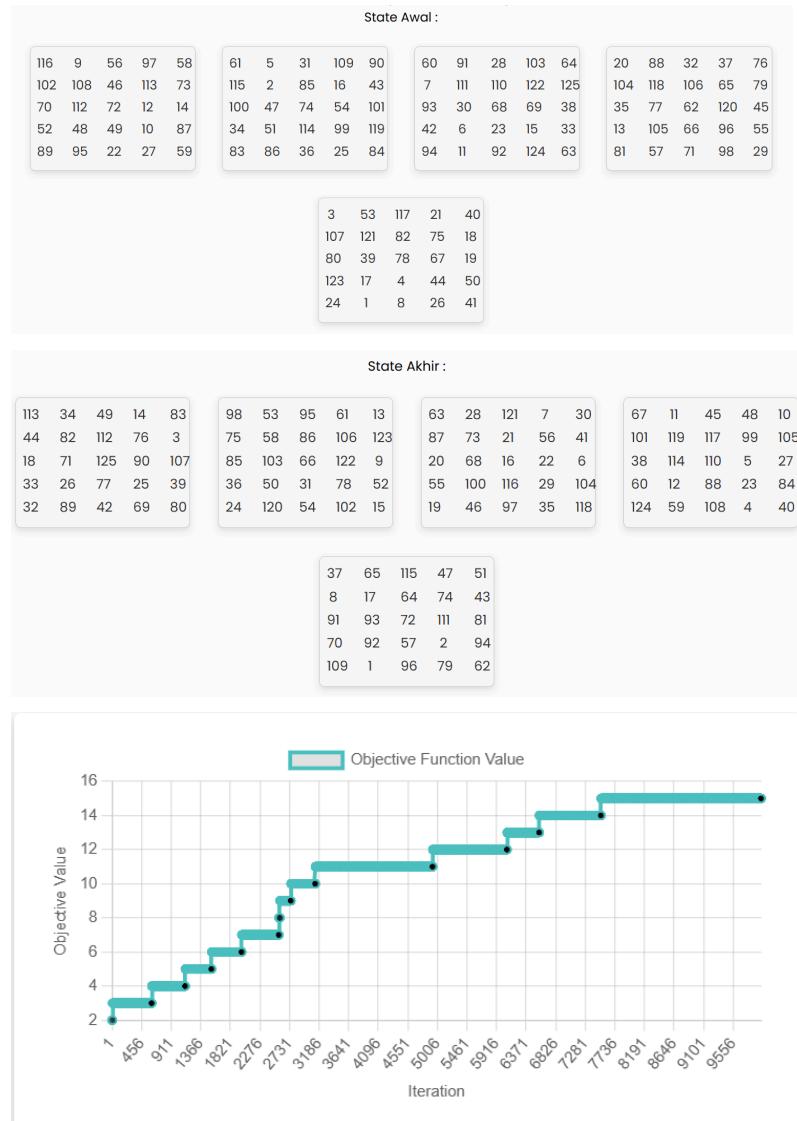
- Variabel kontrol : maxPop dengan maxPop = 10000
  1. Percobaan 1

**Final Score: 19**  
**Iterations: 9990**  
**Duration: 53.109 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 10000**



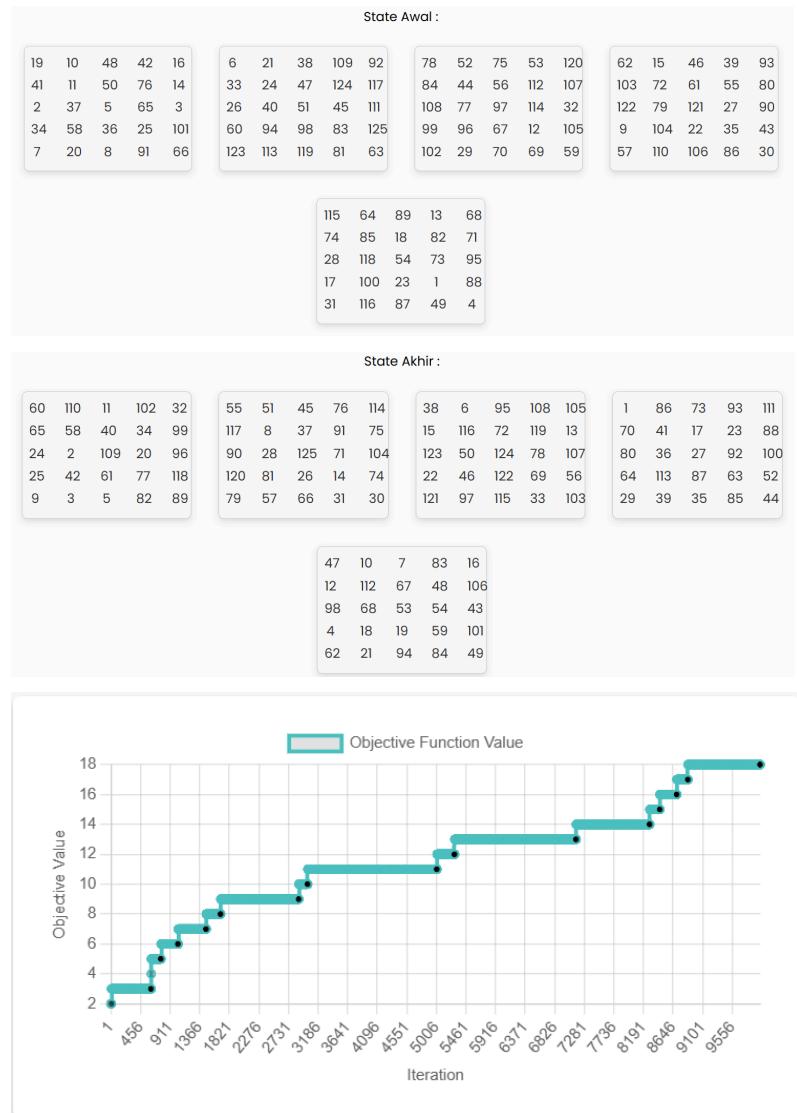
## 2. Percobaan 2

**Final Score: 15**  
**Iterations: 9990**  
**Duration: 53.052 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 10000**



### 3. Percobaan 3

**Final Score: 18**  
**Iterations: 9990**  
**Duration: 54.553 seconds**  
**Sum of obj: 0**  
**Total Populasi (Khusus Genetic) : 10000**



Berdasarkan hasil seluruh percobaan tersebut, dapat disimpulkan bahwa algoritma *Genetic Algorithm* mampu terus meningkatkan nilai fitness function terbaik dari sekumpulan individu pada populasi tersebut. Selama algoritma berlangsung, fitness function atau objective function terus meningkat karena mengambil individu terbaik sebagai parent sehingga dapat menghasilkan objective function akhir yang baik. Namun dengan catatan durasi dan jumlah iterasi yang dibutuhkan sangat banyak.

Apabila dibandingkan dengan algoritma lain, dengan jumlah iterasi atau durasi yang sama algoritma ini hanya mampu menghasilkan value

objective function yang kecil. Waktu yang dibutuhkan dalam melakukan suatu iterasi sangatlah banyak pada algoritma ini. Namun konsistensi algoritma ini relatif baik dengan perbedaan yang tidak signifikan dengan variabel kontrol yang sama dan objective function juga meningkat perlahan apabila variabel kontrol dinaikkan. Jadi, algoritma ini dapat dekat dengan solusi akhir namun dengan catatan tidak mempertimbangkan durasi ataupun jumlah iterasi. Dengan kata lain, semakin tinggi durasi dan jumlah iterasi, semakin dekat value objective function ke solusi optimal.

### III. Kesimpulan dan Saran

- Algoritma steepest ascent hill climbing mampu menghasilkan nilai objective function yang optimal dengan durasi yang singkat, namun rentan terjebak di kondisi local optimum.
- Algoritma stochastic hill climbing menghasilkan nilai objective function yang optimal, namun durasi yang dibutuhkan lebih lama dan masih rentan untuk terjebak di kondisi local optimum.
- Algoritma simulated annealing menghasilkan nilai objective function yang baik dengan waktu yang singkat, algoritma ini dapat menghindari kondisi local optimum.
- Algoritma genetika menghasilkan nilai objective function yang baik, namun durasi yang dibutuhkan lama hal ini karena algoritma ini selalu mengambil parent terbaik untuk dilakukan crossover dan akan menjadikan child sebagai parent apabila memiliki fitness function yang lebih baik sehingga selama algoritma berlangsung fitness function dari setiap kondisi populasi akan terus meningkat.
- Berdasarkan evaluasi keempat algoritma, **Simulated Annealing** dapat disimpulkan sebagai metode terbaik karena mampu menghasilkan nilai **objective function** yang baik dalam waktu yang relatif singkat, serta mampu menghindari jebakan **local optimum**. Meskipun **Algoritma Genetika** juga memberikan hasil yang optimum dan konsisten, **durasi eksekusinya yang panjang** menjadi kelemahan signifikan dari **algoritma Genetika**. Oleh karena itu, dalam konteks efisiensi waktu dan kemampuan mengatasi jebakan solusi lokal, **Simulated**

**Annealing** adalah algoritma yang paling unggul untuk mencapai solusi yang optimal.

- Disarankan untuk pemecahan permasalahan magic cube untuk menggunakan **algoritma Simulated Annealing** dengan alasan berdasarkan kesimpulan diatas.

#### IV. Pembagian tugas tiap anggota kelompok

NIM	Nama	Pembagian Kerja
18222108	Muhammad Daffa Kusuma A	<ul style="list-style-type: none"><li>• Membuat User Interface web</li><li>• Membuat implementasi simulated annealing</li><li>• Membuat dokumen bagian simulated annealing</li></ul>
18222117	Ahmad Fawwazi	<ul style="list-style-type: none"><li>• Membuat objective function dan fungsi display</li><li>• Membuat implementasi steepest ascent hill climbing</li><li>• Membuat dokumen bagian deskripsi persoalan, objective function, dan steepest ascent hill climbing.</li></ul>
18222129	Muhammad Faishal Putra	<ul style="list-style-type: none"><li>• Membuat fungsi cursorgrab</li><li>• Membuat implementasi stochastic hill climbing</li><li>• Membuat dokumen bagian stochastic hill climbing dan kesimpulan saran</li></ul>
18222136	Muhammad Faishal Firdaus	<ul style="list-style-type: none"><li>• Membuat struktur algoritma visualisasi kubus</li><li>• Membuat implementasi genetic algorithm</li><li>• membuat dokumen bagian genetic algorithm</li></ul>

## V. Referensi

1. Trump, W. Perfect Magic Cubes. Diambil dari [Perfect Magic Cubes \(trump.de\)](http://Perfect%20Magic%20Cubes%20(trump.de))
2. Scaler Topics. local Search Algorithm In Artificial Intelligence. Diambil dari [local Search Algorithm In Artificial Intelligence - Scaler Topics](http://local%20Search%20Algorithm%20In%20Artificial%20Intelligence%20-%20Scaler%20Topics)
3. Magisch vierkant. 5x5x5 magic square. Diambil dari [5x5x5 magic square - Magisch vierkant](http://5x5x5%20magic%20square%20-%20Magisch%20vierkant)
4. [Reading 03: Complete Search, Greedy Algorithms \(nd.edu\)](#)
5. <https://www.youtube.com/watch?v=MCVkMmYL-aY&list=PLFIM0718LjIWaNi4oDdN49FTlznrWU9w2>
6. [Features of the magic cube - Magisch vierkant](#)
7. [Perfect Magic Cubes](#)
8. [Magic cube - Wikipedia](#)